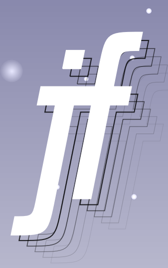
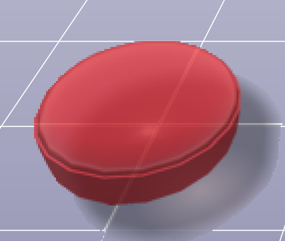


user manual

The Red Pill

personal edition

version 1.0 (October 2, 2010)



Contents

1 English

1.1 Introduction

The Red Pill, further just *TRP* is a project started with an idea to make a device that is able to change the avatar. The project name comes from the pill, Neo took in the film Matrix. After Neo took the red pill, his life changed completelly. Something similar should do a red pill in SL.

It should be an attachment transforming the avatar. If to a cat, a bird, a robot, or a cloud of fire. Step by step, with chat messages and running animations. Within minutes or even hours, automatically and immersive. But more important was not to develop a device for a special transformation, but to create a technology for development of such devices.

To acheive this goals, the programming of the transformation was separated from the performing it: The process is writen into a notecard as a sequence of commands. This sequence is read and executed by a script. The commands are easy to learn and allow to create new transformation scenarios or adapt existing ones.

Thus, TRP is an interpreter, that reads the program notecard and executes the commands in it. Thereby it allows to develop devices running time-controlled actions, like opening doors, saing chat messages, switching light chains on and off and more. In this document we will see the development of a drink glass while the drinking the content and taking off the glass is controlled by TRP.

Brief overwiev of TRP commands

- Chat commands: shouting, saying, whispering, private chat messages
- Flow control: `if`, `for`, `while`
- User interaction via dialog message
- Device control and interaction via linked message
- Adding, removing RLV restrictions and enforcing actions
- Changing avatar body parts, clothes and attachments
- Checking of using RLV and wearing clothing parts or attachments

1.1.1 Package content

This content is inside the poduct box, picture 1.2.

- Scripts `trp.core`, `trp.rlv` and `trp.emergence`
- License notecards `JFS Developer License Agreement`, `JFS Entwickler Lizenzvereinbarung`
- Help notecard `TRP SE Help` with embedded notecards in english and german: User manual and tutorial, language and API specifications
- Tutorial items: object `waterglass`, notecard `autorun.trp` and (driver) script `waterglass.drv`
- Landmark to the JFS main store
- Script '`*productKey`' for product updates and Keyholder device, already load with it

1.1.1.1 Script permissions (builder edition)

The scripts '`trp.core`' and '`trp.rlv`' are sold with permisions *copy, no-mod, transfer*, figure 1.2. As long the permissions remain so, the scripts are not runable. To 'activate' them, the next owner permissions of the scripts should be reduced to either *no-copy* or *no-transfer*. This will not prevent you from distributing your creations running TRP, and also not prevent you from setting any inworld next owner permisions on them.

1.1.2 TRP architecture

The TRP architecture is shown in the figure 1.1.

The architecture is build off four layers. The upper layer is a *TRP program* contained inside a notecard. Actually, every program name is possible, not just '`program`', but the name of notecard containing it must be the program name extended by '`.trp`'. The name '`autorun.trp`' is special: A notecard with this name is started automatically.

The second layer is the *TRP interpreter*, build off the interpreter script (or core script) '`trp.core`' and several plugin scripts. A plugin is a device-independ script, required for executing a number of program commands or supporting the intepreter and other plugins. Plugin scripts should use the name prefix '`trp.`'

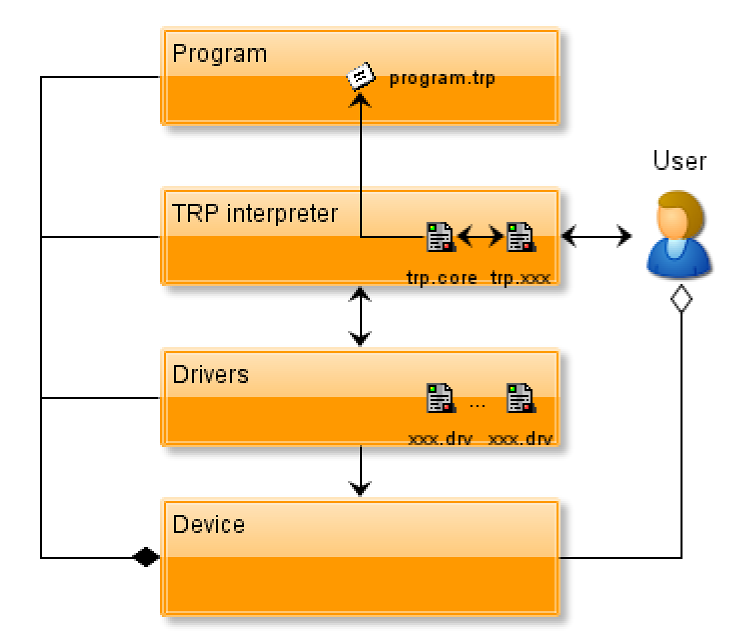


Figure 1.1: TRP Architecture

The third layer make the *device drivers*. Device drivers are device-depend scripts required for the interpreter-device interaction. The drivers are not part of the TRP but a part of the TRP installation. Driver scripts should use the name suffix '`.drv`'.

The lower layer is the device itself. It belongs to the user and contains physically the program notecard, the TRP interpreter scripts and the driver scripts.

1.1.3 Immersion and RLV

The avatar transformation is simple changing the avatar shape, skin, attachments and an AO, but doing it manually leaves less room for immersive experience. Also, by using the standard viewer, the resident has all controll over their avatar every time while the transformation process. This ruins the feeling being transformed.

Fortunatelly, there are several so-called RLV-compatible viewers in SL, section 1.1.3.3. Actually *RLV* is the name of the first viewer that extended the regular SL viewer by an 'adult' ability. This ability was defined and released as an API[1] and now implemented by other viewers, [9]. For this reason viewers implementing this API are called RLV-compatible or just RLV. The API itself is addressed often by *RLVa*, thus also here.

Why is this API important for TRP? Commonly, an API is an interface between machines or programs. A program like viewer offers this way a possibility to control

[1] Application programing interface

it to other programs like scripts. This allows the TRP interpreter to access to sertain functions of the viewer to improve the resident's immersive experience.

Running of RLV-compatible viewers is not required to use TRP devices. Actually the code, accessing this viewer is collectig into the '`trp.rlv`' sctipt, the *RLV plugin*. If this script is not installed, the device might work stil, but without accessing the API of the viewer.

1.1.3.1 This the RLVa allows to do (a really few examples)

- Accepting of items like shape, skin, attachments, clothing into a special folder in avatar inventory
- Automatical wearing of items in this folder
- Rendering worn attachments or clothing non-changeable
- Automatical removing attachmens or clothing
- Restricting the ability to fly or touch far objects
- Blocking acces to inventory, editing or rezzzing objects
- Blocking IM and chat conversations
- Hiding maps, actual location and avatar names
- Restricting and forcing windlight environment settngs
- Automatical gridwide teleport like using a landmark

1.1.3.2 This is NOT possible to do by RLVa (for example)

- Deleting items in inventory or giving them away
- Giving the money away, without the user knows or allows it
- Revealing the private IM conversations
- Stealing passwords or credit card data[2]

[2] At least popular viewers not do such nasty things

1.1.3.3 List of (some) RLV-compatible viewers

Trere are for sure more, but this few are popular viewers, known to the author[3] and known to have the RLVa implemented:

- Classical RLV, [8]
- Imprudence, [3]
- Cool Viewer, [1]
- Rainbow Viewer, [10]
- Phoenix viewer, [7]
- Emerald Viewer, [2]

This viewers are third-party viewers, that means they are not released by the labs. To encourage the developers to create confidential viewers, the labs released a third party policy (*TPVP*, [12]) and a directory for third-party viewers (*TPVD*, [11]). The TPVD lists viewers, their developers self-commінted to the policy.

The first two viewers, the RLV and Imprudence, you find on the TPVD, that means the develoers assure, their viewers are safe.

The Cool Viewer and Rainbow Viewer were never on the directory, the reasons you can read on developer's pages. Both viewers were thesame a while ago, than they disjointed the developmet, the stories about that you can find in the on thesame pages, too, for sure. The Cool Viewer is stil in development at time, the Rainbow no more.

The Emerald Viewer was on the directory once, but was taken from it after comung up of events breaking some terms of the policy and lost the permissin to access SL grid. It is no more in development. The viewer was a great development, it was very popular in the past, and will give derivates, too. One of the actual derivates is the Phoenix viewer.

1.1.4 No-script areas

Perhaps you know already that areas where everything stops working, except vehicles and perhaps AOs, because scripts are not allowed there to run. Well, for many things this setting is just annoying, can lead to trapping situations if the session uses RLV (from view of the avatar).

[3] Jenna Felton

Such situations are those, the avatar can't escape without help. Before we start discussing if such sutuations are possible or allowed at all, imagine a transformation of a pupa into a butterfly.

The pupa can do nothing but await the end of transformations. If an avatar is in such transformation, it may not fly, not teleport and should have no possibility to cheat around this restrictions, in order the player feels really being a larva.

After, say, 10 minutes wings appear and all restrictions are gone. Usually. Now, the avatar hapens to come, occasionally or intentionally into a no-script area while this transformation. 10 minutes become an endless eternity.

To prevent such situations is there the '`trp.emergence`' sctipt, the *emergence plugin*. It uses a trick and pretends being a vehicle: It takes the avatar's control. Vehicle scripts are allowed exceptionally to run in no-script areas. Some AOs work for this reson in no-skript areas too, they take also the avatar's control.

However, this plugin should be used sparelly and installed only if really necessary. There are some good reasons not to take it's code into the interpreter script. At many places the scripts are disallowed not for fun. The owner could want save server resources for vehicles or avatar movement.

The script itself takes also resources to run. There is a timer that checks periodically if the control is stil taken and if not does so. And taking control means as soon arrow keys are pressed, some additional code is executed by the script engine. Well, the script was written with a goal to be lag-friendly: The timer rate is set to 15 seconds (relativelly low) and the control is taken over the page-down key, in the hope that the sim will execute not much code, as the key is rarelly used. And the plugin is only active, if any TRP program is executing, and just waiting at other time.

BTW, if a standard viewer is used, the button 'Release keys' appears as soon the plugin becomes active. After clicking it away it appears again in 15 seconds. Very anoying if no device is used that seems to use the control, like vehicles. Only the RLV (and compatible viewer) have learned to hide the button. Means: If the RLV plugin is not used, you not need the emergence plugin, too.

And finally, the emergence plugin can be omitted if the device you equip by TRP takes control anyway, i.e. it is a vehicle or some AO script. Than the plugin would waste resources for nothing.

1.1.5 Release notes

v1.0 (October 2010)

- Initial release

1.2 JFS Developer License Agreement

DLA, version 1.2

By purchasing of this Product you acept the tems of this license agreement.

Subject of the agreement

1.1. The license aggreement rules distribution of JFS development products ("*product*") and creations developed by using it ("*creation*").

1.2. Distribution means giving out the product or parts of it, in any means, regardless of content being given with it.

1.3. The product distribution must contain information about its source, and if the product content was changed, also name the changes taken place. This term is only out of force, if satisfying it would break the distribution process.

1.4. The product distribution must contain an unchanged copy of this license aggreement. This term is only out of force, if satisfying it would break the distribution process.

Full-perm product content

2.1. The full-perm product content, regardless of the content type, should remain full-perm while distribution, regardless if distributed as is or together with other creations or inside them.

2.2. The content as subject of the term 2.1. should be distributed for free if the content was unchanged or experienced minor changes and not coming with parts of product content matching the terms 2.3., 3.2. and 3.3.

2.3. The content as subject of the term 2.1. might be distributed for any prize only after significant modifications on the whole content or important parts of it.

No-mod product content

3.1. Product content without the owner's mod permission, regardless of the content type except notecards, may not be distributed if the next owner's permissions remains *transfer and copy*. This term applies to the content distributed as is or together with other creations or inside them.

3.2. If the content as subject of the term 3.1. is a primary part of distribution process, the item being distributed may not be given for lower price as the adequate product of JFS.

3.3. If the content as subject of the term 3.1. is not a primary part of distribution process, the content or the item being distributed may be given for any prize.

In other words...

The TRP package contains full-perm notecards, landmarks, textures and objects with further content. These and the full perm content of the objects must remain ful-perm. And they can be only given out for free, as long they stay not changed or almost unchanged (experienced *minor* changes).

Minor changes mean for example renaming items, removing typos from script comments or notecard text, without changing the text meaning. Or renaming variables in the script or exchanging the script lines without altering the script functionality. Such changes are not important enough to be seen as *significant* changes, doing them you may not claim prize for redistributing the changed product.

Significant changes are e.g. graphical improvements of objects, changing features of scripts, or adding own scripts or documentations, affect the whole product or parts of it. Changed so, you may distribute the product with any price. Furthermore, you must not sell the changed part and give the unchanged part of the product for free separatelly, changed contents override the unchanged contents of the product in same package.

What the other part of the product you sell is out of interest for this DLA, but if you distribute it, you have to add some information about TRP, and document also all changes you made on the TRP content, for example scripts or notecards, and you must put a copy of this license aggreement into the package you sell.

There is an exception allowing you not to provide the source informations and the license copy: If you sell a running device, like a drink attachment runing the TRP and you give out the device by a vending machine. Giving the informations and licene notecard together with the drink will break the vending process, thus you can skip it.

This license aggreement aims to ensure that TRP (as every other JFS development products) becomes no freeby and on the other side to allow you a free development and distribution of products running TRP.

1.3 Application of TRP

1.3.1 Enpacking

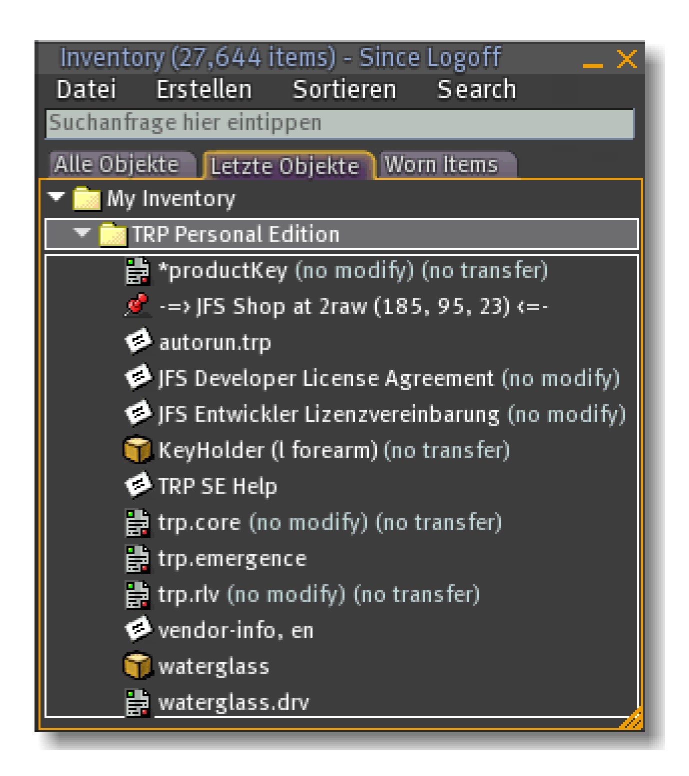


Figure 1.2: Content of the product package

The easiest way to unpack the product box is to use the box itself, if you do so you'll not have to care about the product box left in the world. So, please look for a place where you can rez stuff and that allows scripts to run. Best choice is your own home or a sandbox.[4]

Now rezz the product box. A menu will pop down, please hit the *Open* button. The button *Update* you'll need to take an update from the update orb and the button *Remove* will remove the box from the world what is too early at this step.

After a folder is given to you (accept it please) another menu appears, please hit the *Remove* button now. This will destroy the rezzed box. If you ignore this (or the prevous) menu, you can touch the product box again to make it open the first menu where you can use the *Remove* buton, too. If you don't do so, after 15 minutes a message will remind you about forgoten box and if you stil not react, the box will delete itself after further 15 minutes.

[4]If a place allows scripts to run you find out in the land info.

That's all, the content of the product box you'll find as a folder in your inventory, figure 1.2. Please do not remove the product box from your inventory, it is a good backup and also a way to request product updates.

1.3.2 Installation

We introduce the installation of TRP system by using the supplied tutorial items, section 1.1.1. The instalation and adjstment of a TRP system should go in this order

1. The *system device*, in our case the '`waterglass`' item. Please rezz it on the ground, edit it and switch to the *Contents* tab.
2. All *wearable items* for the transformations go into device inventory. That are body parts like shape, skin, eyes, clothing and attachments. In our case there are no such items.
3. All *driver scripts* go into device inventory: Only the '`waterglass.drv`' script in our case.
4. All *plugin scripts* go into device inventory. Plugin scripts are '`trp.rlv`', '`trp.emergence`', but they are not needed in our case.
5. The *interpreter script* '`trp.core`' go into device intentory too. We take a version with appropriate permissions, for example one from *no transfer* folder.
6. All *programs*, i.e. notecards with name suffix '`.trp`'. In our case it is the only notecard '`autorun.trp`'.

Note: In this order, the interpreter script is initialized before the '`autorun.trp`' is placed. This way the interpreter starts the execution not immediatelly but after you pick up the device and rez or wear it next time.

If you want start executing the notecard '`autorun.trp`' immediatelly while the installation, install please the notecard after other program notecards but before the interpreter script. The steps 5 and 6 you exchange than please.

1.3.3 Updating

The JFS updater uses a special update system. Earlier it was included into each JFS vendor, for now it is installed inworld as an update orb separatelly. The orb communicates with a '`*productKey`' script, supplied with each JFS product box. The script is able to generate a valide update request for the accordant product. It is actually the only possibility to take a product update: Only if you bought a

product you have the '*productKey' script, and only if you have it, you can request updates for the product.

The '*productKey' script is set up in the product box and running, hence if you rez the box near the update orb, you can request the update for the product immediately. To make it more comfortable and to avoid caring boxes, each for another JFS product, there is a key holder device, sold for free in the JFS vendor and also containing in the product box. The key holder manages up to 12 product keys, i.e. by wearing the keyholder you can request updates for up to 12 products. But again, you must stand close to the update orb than.

In any case, to request a TRP update you must be close to the update orb, instlalled inworld. At time this document was written, the update orb was installed at main store, [6]. Since the list of installation locations changes often, a blog post was created to list the actual JFS shop locations, and to note which of them has update orb installed as well, [4].

Note: The update is just a redeliver of the actual product package. After receiving it you might repeat the enpacking and preparing processes again, section **??**.

1.3.3.1 Updating via product box

If you use the product box or any prim with '*productKey' script installed:

1. Please come close to the update orb (less than 10m)
2. Rez the (product) box, or wear it, a menu appears
   a) If not, touch the box for menu please
   b) In the menu is an *Update* button, klick it
   c) The update orb will send you a copy of the actual TRP package

1.3.3.2 Updating via Keyholder

If you use the keyholder device:

1. Please come close to the update orb (less than 10m)
2. Wear the key holder device and touch it please, a menu appears
   a) If the TRP key is not selected, please select it
   b) Now hit the *Update* button
   c) The update orb will send you a copy of the actual TRP package

That's all than :)

1.4 Program example

```
1  # TRP-controlled device: Glass of Mineral Water
2  # Scenario: * intro
3  #           * drinking until empty
4  #           * interactive unwearing
5  #
6  # Version 1.0, $Revision: 1.3 $$Date: 2010/09/30 20:30:43 $
7
8  rename %fn%' glass
9
10 wait 3
11
12 whisper %an% takes a glass of cool drink
13
14 while not lm 15 this 0 isempty?
15     lm this 0 drink
16     wait 10+15
17 end
18
19 whisper //me is empty
20
21 if dialog 30 Detach, Drop, Place; What to do with empty glas?
22     lm this 0 detachme
23 else
24     whisper %fn% drops the glass on the floor
25     lm this 0 dropme
26 else
27     whisper %fn% puts the empty glass back on the table
28     lm this 0 placeme
29 default
30     lm this 0 detachme
31 end
32
33 clean script
```

Figure 1.3: TRP program, supplied example notecard

1.5 TRP programming language

TRP is a language interpreter in first place. This section will introduce the TRP programming language[5]. To do it, we'll take a brief look at a ready program first, notice it's syntactical properties and than introduce each TRP command in detail.

1.5.1 Syntax

We investigate the supplied example notecard, figure 1.3.

1.5.1.1 General attributes

- The char '`#`' and the text until the line end is a comment
- All other words are *commands* with none, one or multple *parameters*
- Secondary commands of a large construct are called *operators*
- Command parameters are mostly space-separated and never taken into brakets
- Each command must use a separate line, e.g. there is no `else if` operator
- The *line indention* has no relevance for the interpreter but improves readability
- Command names are *case-insensitive*, `else` and `ELSE` mean thesame
- There are no variables or user-defined functions

1.5.1.2 Code explanation

Let's go the example 1.3 line for line.

The short comments block (lines 1 to 6) is an usual preamble, introducing the code, there is nothing to explain. Just one notice: Empty lines and lines with comment code only take script time for reading but do nothing. In real programs, better to avoid such lines: Place the comments after commands in same lines and put code lines after each other. This practice will reduce lag, but for sure also the code readability.

The line 8 tells the interpreter to rename the device by using the owner's first name. If Jenna Felton takes the glass, than it's name will become "*Jenna's glass*".

The `wait` command in the line 10, tells the interpreter to make a stop for exactly 3 seconds (to give others viewers a time to rezz the glass).

[5]Sharp tongues could call it rather a scripting language

The `whisper` command in the line 12 tells the interpreter to whisper the given text, while this time the owner's full name is used. If, again, Jenna Felton takes the glass, than this command will whisper "*Jenna Felton takes a glass of cool drink*".

Now comes a first interactive part, the `while` statement at lines 14 to 17. The `while` command takes the condition "`lm 15 this 0 isempty?`" as parameter and executes two lines of code (lines 15 and 16) as long the condition is not true.

The condition means this: Send a command "`isempty?`" as a linked message to the device driver and await it's answer. What the meaning of the command is, decides the driver. In this case it is a question if the glass is empty or not yet. As long the answer is negative, the glass is not empty yet and the loop code has to be executed.

The loop code are the lines 15 and 16. The line 15 tells the interpret to send the driver a command "`drink`" as a linked message (the driver would understand is as command to run a drinking code) and the line 16 tells to wait a random time between 10 and 25 seconds. After it the loop condition will be checked again.

As soon the glass becomes empty, the `while` loop is left and the TPR interpreter finds the `whisper` command at line 19, reporting the glass is empty. The double slash results in a single slash in the message making an object emote.

Now the interpreter finds another interactive part, this time a user interaction: The `if` statement at lines 21 to 31. First, the interpreter finds the `if` operator telling to check the condition and decide what to do. The condition is a `dialog` operator. As result the interpreter opens a dialog with three buttons '*Detach*', '*Drop*', '*Place*' and the question. The number 30 means, the interprete should give the user 30 seconds to answer the dialog.

Well, the user can hit the first, second or third button, or ignore the dialog. Depends on it, the interpretor runs the code before the first `else` operator, before the second `else` operator, before the `default` operator or after it. The code in selected branches tells the interpretor to send one of the commands "`detachme`", "`dropme`", or "`placeme`" via linked message, so the driver could perform the action, and in two cases the taken action is also commented in chat.

Regardless what case was executed, the interpreter continues with code after the line 31, the `end` operator. There is only the last command there at line 33, the `clean` command with `script` parameter. It tells the interpreter to stop the program execution and to remove the TRP scripts from the glass inventory, making it an one-way drink.

1.5.1.3 Syntax templates

To introduce each TRP command easier, syntax templates were used in this documentation. There are four of them in use: Command placeholders, optional ele-

ments, element lists and enumerations.

1.5.1.4 Command placeholders

```
1 think <text>
```

Command placeholders are taken into acute brakets and is replaced in a real command with another text. What exactly can replace the placeholder is the decision of semantics of the command `think`. In this case its a chat command, thus this command matches the definition for example:

```
1 think Hello World!
```

1.5.1.5 Optional elements

```
1 whisper [/<channel>] <text>
```

Square brakets denote optional command parts, whch can be omitted. in the example above two placeholders are used, an optional `channel` and mandatory `text`. The slash belongs to the channel placeholder and must be omitted if the channel is not used. This two commands match the definition:

```
1 whisper Hello World!
2 whisper /100 Hello World!
```

In the line 1 the channel is omitted, in the line 2 the channel is present.

1.5.1.6 Lists of Elements

```
1 (, <button> )+
```

A list of similar elements is given by a round brakets, a separator char and a quantor char. In the example above a list of placeholders is defined, possibly buttons, the separator char is a comma and the quantor char is the plus. This quantor means, the list must have at least one button. This list matches the definition “`Apple, Orange, Plum`”.

The separator char shall stand only between the list elements. The plus char is the only separator char used in this documentation. In common use are also the asterisk ‘`*`’ (the list may be empty) and the question mark ‘`?`’ (the list can be empty or have only a single element), this definitions can be replaced by an optional list or optional element.

1.5.1.7 Enumeration

```
1 command (value1 | value2 | value3) option
```

Enumerations define a selection from a given list. This list is enclosed in round brakets with a vertical bar as separator char and without a quantor char. The exmple definition resolves for example to the command “`command value2 option`”.

1.5.1.8 Space rule

Usually you can set space chars around elements, than the definition uses them too, like this:

```
1 say [/<channel>] <text>
```

The command “`say /100 Welcome`” matches the definition and “`say      /100    Welcome`” does as well. If space chars are not allowed in the command, they are also omitted in the definition, like this:

```
1 @(,<command>[:<option>[=<parameter>]])+
```

The string “`@cmd:opt=par,cmd2:opt2=par2`” is allowed by the definition and the string “`@ cmd:opt=par, cmd2 : opt2=par2`” is not.

1.5.2 Core commands

Core commands are single-line commands, interpreted by the core script.

1.5.2.1 Identity command: RENAME

```
1 rename [<name>]
```

The `rename` command gives the device another name for chat messages. If the parameter is omitted, the name is restored to the original device name. The text supports also text placeholders, table 1.1. For example:

```
1 rename %fn%'s glass
```

If the glass or water belongs to Jenna Felton, its renamed to “*Jenna’s glass*”.

`%an%`	is replaced by the full name of the owner, e.g. *Jenna Felton*
`%fn%`	is replaced by the first name of the owner, e.g. *Jenna*
`%ln%`	is replaced by the first name of the owner, e.g. *Felton*
`\n`	is replaced by line break (not recomended for names)

Table 1.1: Text placeholders

1.5.2.2 Chat commands: SHOUT, SAY, WHISPER, THINK, IM

```
1 shout [/<channel>] <text>
2 say [/<channel>] <text>
3 whisper [/<channel>] <text>
4 think <text>
5 im <text>
```

TRP supports this five chatting commands. All them support text placeholders for the `text` parameter, table 1.1. The commands `shout`, `say` and `whisper` correspond to shout, say and whispering messages on the chat line, they come just from an object and are dark green in chat history. The commands allow to name the channel for sent messages. Negative channel numers are also possible:

```
1 shout Hello there!!!
2 say /100 can you hear me?
3 whisper /-23 no, you can't :)
```

To start the text with a slash, and if no channel number is used, a double slash should be used, as the single slasch will mean to take the first part as channel number and this wil fail. Example:

```
1 say //me is empty. %fn% feels much better now...
```

The command `think` generates a private owner message. It is yellow in chat histroy and bursts chat particles. The sommand `im` sends an IM to the owner. This message is light green in chat history and produces no chat particles but delays the intrpreter for 2 seconds.

1.5.2.3 Dialog message: MESSAGE

```
1 message <text>
```

The command `message` shows the owner a blue message box with given message and the only *Ok* button. This commands neither delays, nor creates any listener. The dialog button has no effect. The message text supports text placeholders, table 1.1.

Although this message is fast and takes the user's attention like nothing else, it is advisable not to use it too often: The user could loose the overwiev about messages since the latest dialog message is shown first. Example:

```
1 message WARNING\nPlease read my messages carefully!
```

1.5.2.4 Link message: LM

```
1 lm (<link> | root | this | all) <number> <text>
```

The `lm` command sends a linked messag to the device driver. In the example 1.3 this command was used to enforce the drinking and detaching actions of the driver.

The first parameter determines the message target. Either a number, than it is the prim number in linkset[6], or one of three names: '`root`' to send the message towards the root prim, '`this`' to let the message stay in the prim where the interpreter is runing, and '`all`' to target the entire linkset.

The parameter `number` is the nummeric parametr of the linked message. Any number except `-2018160314` is possible. The parameter `text` is the string parameter of the linked message. This parameter supports also text placeholders, table 1.1. The key parameter of the message is always `NULL_KEY`. Example:

```
1 lm this 0 take drink
```

The effect of this command will be executing this LSL command:

```
llMessageLinked(LINK_THIS, 0, "take drink", NULL_KEY);
```

Note: Please notice the difference: The command `im` sends an *instant* message, the command `lm` a *linked* message.

1.5.2.5 Pause command: WAIT

```
1 wait [[[<day>:]<hour>:]<min>:]<sec>
```

This command stops the execution for a given amount of time. The time is given by days, hours, munutes and seconds. You can omit them from days to minutes. Each component can be given a fixed number by using a number or a random number by usind a plus operator:

[6] The root prim in a linkset has the number 1, other prims have larger number

```
1 wait 40            # wait 40 sec
2 wait 3:40          # 3 min, 40 sec
3 wait 12:3:40       # 12 hours, 3 min, 40 sec
4 wait 1:12:3:40     # 1 day, 12 hours, 3 min, 40 sec
5 wait 10+30         # wait between 10 and 40 sec
6 wait 10+2:3:10+30  # between 10 hours, 3 min, 10 sec
7                    # and 12 hours, 3 min, 40 sec
```

1.5.2.6 Context switch: START

```
1 start <name>
```

This command takes a program name and switches to it's execution. The interpreter looks than for a notecard named '`<name>.trp`'. Is it found and checked to be *executable*, the interpreter starts executing the notecard. For example, this command line forces execution switch to the '`strawberry.trp`' notecard:

```
1 start strawberry
```

- A notecard is executable, if it has content and is full-perm in device inventory.
- After execution of the target program, there is *no return* to the program where the switch was triggered.
- An executable program '`autorun`' is started automatically:
  - after reset of TRP scripts
  - if the device changes the owner
  - while rezzing if the program is not running

1.5.2.7 Termination command: CLEAN

```
1 clean [script]
```

This command `clean` terminates the program execution.

- If no parameter is used, the program just stops.
- If the parameter `script` is used, the execution stops and the TRP interpreter and plugin scripts are removed from the device inventory.

Using of the paramter makes the device not resetable.

1.5.3 Flow control

Commands of this group execute command blocks in dependancy of defined conditions. There are three commands: `if`, `while` and `for`.

1.5.3.1 IF constraint

```
1 if [not] <condition>
2     # commands #
3 else
4 ...
5 else
6     # commands #
7 default
8     # commands #
9 end
```

The `if` constraint allows to select a command block for execution in dependancy on a certain condition, while the condition is given by command parameter.

1.5.3.2 Handling conditions in TRP

Popular programming languages handle the `if` command as a selection of branch to execute while the condition can take two posibilities, either it is true or false, e.g. the statement $x < 4$ can be either true or false. Many programming languages know also another selection operation, where a value of a given variable is compared with a given list of values and depends on the result one of many branches is executed.

However, the script memory in LSL is not unlimited, hence the TRP `if` constraint covers both operations. To acheive it, conditions in TRP are always resolved to the number of branch to be executed. This number gives the count of `else` operators to skip until the wanted branch is found.

That means: If the condition is true, it resolves to the branch number 0. The interpreter executes commands, following the `if` operator until first `else` operator is found. If the condition is false, it resolves to the branch number 1. The interpreter executes the branch following the first `else` operator. Conditions that handle more than two cases can resolve to a larger number than 1 and the interreter must skip more than one `else` operator.

The `default` operator starts an operation branch to be only executed if the interpreter was unable to resolve the nuber of branch for executon. This hapens for example if the condition was about user selection and the answer not came in time.

1.5.3.3 Negated condition

The `not` operator negates the condition, i.e. makes *true* to *false* and vice versa. Since the TRP conditions are resolved to numbers, the operator just converts the numbers by substracting from 1:

- The *true* condition means 0. After the `not` operator it becomes 1, thus *false*
- The *false* condition means 1. After the `not` operator it becomes 0, thus *true*
- A condition that resolves to larger number than 1 becomes negative and the interpreter finds no branch to handle it
- The `not` operator has no effect to executability of the `default` branch

1.5.3.4 Random numbers: RND

```
1 (if | while) rnd <num>
```

The `rnd` condition takes a number between 0 and 100 as parameter, rolls a dice[7] and decides for the case 0 (true) or 1 (false). The given number is the probability in percent for the case 0. This condition always resolwes to the numbers 0 and 1.

For example to whisper ‘Apple’ in 3 cases from 10 (30 percent probability) and ‘Orange’ in 7 cases from 10 (70 percent), we write this:

```
1 if rnd 30
2     whisper Apple
3 else
4     whisper Orange
5 end
```

Is the probability number smaller 0 or larger 100 used, it will be threated as 0 resp. 100. By using the number 0 the first branch is never executed, by using the number 100 never the second one. Using the `not` operator inverses the case selection.

Say, we wanna whisper the words ‘Apple’, ‘Orange’, ‘Plum’ and ‘Cherry’, all with 25% probability. We only can select under two cases, thus we nest multiple `if`s:

```
1 if rnd 50
2     if rnd 50
3         whisper Apple
4     else
5         whisper Orange
6     end
7 ...
```

[7] Yes, TRP really knows random numbers

```
7  else
8      if rnd 50
9          whisper Plum
10     else
11         whisper Cherry
12     end
13 end
```

1.5.3.5 User interaction: DIALOG

```
1  (if | while) dialog <time> (, <button>)+ ; <text>
```

The `if` condition allows the interaction with user. The condition takes a number between 15 and 60, a comma-separated list of button names, separated via a semi-colon from the text. The text, again, supports text placeholders, table 1.1. Is the number smaller 15 or larger 60, it is corrected to 15 resp. 60.

For resolving the condition, the user is shown a blue menu with named buttons and the text message. The condition is resolved exactly to the number of klicked button.[8] The provided number is the time in seconds the user has to take selection. If this time ran out, the condition values as unresolved. For example we can ask the user to select the drink taste:

```
1  if dialog 45 Apple, Orange, Plum, Cherry; What taste you prefer?
2      whisper Ah, you love Apple
3  else
4      whisper Oh, orange is bitter
5  else
6      whisper Plum is nice, too
7  else
8      whisper mmmh, cherry...
9  default
10     whisper You really can't decide?
11 end
```

This constraint will show a dialog with four buttons and a question and give the user 45 seconds to decide. Notice the use of the `default` operators that catch the dialog timeout?

[8]The first button in the list has the number 0

1.5.3.6 Device interaction: LM

```
1 (if | while) lm <time> <link> <number> <text>
```

The lm condition allows an interaction with the device driver. The syntax extends the syntax of lm command by a time value. The condition sends a linked message by means of remaining parameters and awaits an answer of the device driver. The answer is a linked message using the same nummerical parameter and any string but NULL_KEY as key parameter. The condition resolves to the number provided in the string parameter. If it is empty, the conditon is valued as unresolved.

In this example we let the device driver select the taste of the drink:

```
1  if lm 15 this 10 select:Apple,Orange,Plum,Cherry
2      whisper my taste is Apple
3  else
4      whisper my taste is Orange
5  else
6      whisper my taste is Plum
7  else
8      whisper my taste is Cherry
9  default
10     whisper I seem have no taste
11 end
```

What hapens here. To resolve the condition, sends the interpreter a linked message by executing the LSL command

```
llMessageLinked(LINK_THIS, 10, "select:Apple,Orange,Plum,Cherry", NULL_KEY);
```

Now the interpreter starts the timer for 15 seconds and awaits the driver to send a linked message with 10 as nummerical parameter, and any string but NULL_KEY as key parameter, which will be taken as answer. Say, the driver decides for 'Plum', than it should send the number 2 as answer (the numbers start with 0), i.e. the driver will execute this LSL sommand

```
llMessageLinked(LINK_THIS, 10, "2", "");
```

If this answer not arrive in time (15 seconds), the condition is valued as unresolved.

The time value used in thr lm condition can be 0 or negative, than the interpretor stops until the answer arrival. This alllows interesting usages but could hang up the interpreter if the driver forgets responding.

1.5.3.7 Conditional loop: WHILE

```
1 while [not] <condition>
2     # commands #
3 end
```

The while loop works very similar to one in other programming languages. It executes the commands block as long the condition remains true, i.e. resolves to 0. If the not operator is used, the commands block is executed as long the condition resolves to 1. Example:

```
1 while rnd 75
2     whisper Hello there!
3     wait 2
4 end
```

This code will whisper the greeting with probability of 75%, and if done so, the condition will be checked again and with probability of 75% the device will greet again.

1.5.3.8 Counter-based loop: FOR

```
1 for <times>
2     # commands #
3 end
```

This loop executes the commands block a certain number of times. This number can be an exact number or a random number in a range given by the additional syntax. Short Example:

```
1 for 5
2     whisper You hear me exactly 5 times
3 end
4
5 for 5+3
6     whisper You hear me between 5 and 8 times
7 end
```

1.5.4 Introduction into RLV API (RLVa)

The RLV is a name of a special viewer and now of a number of viewers that support the special API, specified in [9]. Please look there if you want an actual and complete specification. Repeating it here would let the cite not actual, and also double the

size of this document. Hence, here we will take just a brief introduction into the RLVa.

About RLV itself and its development, please read in the development's blog, [5], so far at this place: The goal of developing the RLV was to improve the immersive experience of being in SL. To do that, scripts required more control over certain viewer's functions, than it is possible with the original SL viewer.

Well, scripts are executed on the server, and the viewer is a program running locally. Both programs need to comunicate. The elegance of the RLV protocol is in using a communication channel, LL already implemented: Owner messages towards the viewer and chat messages towards the scripts.

Owner messages[9] are yellow in chat history, the viewer receives them only if they come from owned objects. The standard viewer would display all owned messages. The RLV performs the commands but displays all other messages. An owner message is assumed to be RLV commands if it starts with the '`@`' char. The syntax of RLV commands is simple:

```
@(,<command>[[:<option>]=<param>])+
```

The '`@`' char denotes the start of a RLV command block (it ends with the end of the message). Each command starts with the name, following by optional option and parameter values. There is only one command that omit them.

1.5.4.1 Types of RLV commands

There are four kinds of them:

- Information request: "`@<command>[:<option>]=<channel>`"
- Actions: "`@<name>[:<option>]=force`"
- Restrictions: "`@<name>[:<option>]=add`" or "`@<name>[:<option>]=rem`"
- Clear command: "`@clear`"

1.5.4.2 Information requests

This commands allow the scripts to request certain informations from the viewer. There is a number of such operations, but the TRP language has no commands to perform them directly. If needed, the RLV plugin involves them in background.

To be complete, we handle the version request, other requests work similar. The command is `version`. To execute it, a script does this: First, the script chooses a channel number, say it is 1234, and starts listening on this channel.

[9] Owner messages are sent via `llOwnerSay` command.

Than the script sends the owner message "`@version=1234`". The viewer receives it, understands it is a version request and sends it's version string on the channel 1234.

The script receives it, handles it and closes the listen. If the viewer dies not support the RLVa, it simply displays the yellow message and sends nothing. The script does not receive the info in time and closes the listen with knowledge, the RLVa is not supported.

1.5.4.3 Clear command

This is the only command by now, that neither uses the option, nor the parameter: The effect of this command is removing all RLV restrictions, set by scripts in same object.

The TRP interpreter sends this command twice, once bevore starting a program execution and once after completing it, i.e. while executing the `clean` command, section 1.5.2.7.

There is no TRP command to send it separatelly, but you can use the `think` command to produce owned messages, thus by using this command line you can acheive thesame effect (which is not recomend as the RLV plugin is not involved):

```
1 think @clear
```

1.5.4.4 Actions

```
1 @<name>[:<option>]=force
```

Actions emulate clicking some buttons on the user interface. Executing an action changes properties on the avatar but produces no permanent changes to the viewers functions.

Actions are RLV commands with the `force` parameter. Important actions are listed in the table 1.2. Some actions allow options, what value they take, explains the table 1.5.

How to triigger the RLV action. Say, we want unsit the avatar. The action is '`unsit`' and it has no options. Thus to unsit the avatar the script must just send "`@unsit=force`" as an owned message. The viewer receives it and understands: The script clicks the stand up button.

Another more complex example is the action '`detach[:name]`'. A look into the API reveals: If no option is used, this actions removes every attachment. If the attachment point name is used, the action removes the object attached there. By using this actions a script is able to click the attachments and hit the 'detach' button.

Action	Effect
`sit:<UUID>`	Force sit on an object
`unsit`	Force unsit
`tpto:<X>/<Y>/<Z>`	Force-Teleport the user
`remoutfit[:<part>]`	Force removing clothes
`detach[:name]`	Force removing from attachment point
`attach:<folder1/.../folderN>`	Force attach a shared folder
`attachall:<folder1/.../folderN>`	Same, recursivelly
`detach:<folder>`	Detach a shared folder
`detachall:<folder1/.../folderN>`	Same, recursivelly
`detachme`	Force detach the item sent the command
`setrot:<angle>`	Force rotate the avatar
`setdebug_<setting>:<value>`	Force change a debug setting
`setenv_<setting>:<value>`	Force change an environment setting

Table 1.2: RLV actions and their effect

1.5.4.5 Restrictions

```
1 @<name>[:<option>]=add
2 @<name>[:<option>]=rem
```

A restriction is a permanent viewer state deactivating its certain ability[10]. For better understanding, you can imagine the set of restrictions as a page in the preferences dialog with checkboxes: Disallow fly, hide inventory, hide minimap, and so on. However, this page is only visible to scripts and not to the viewer's user.

Since restrictions are permanent settings, there are two params to manage them. The `add` parameter sets the restriction, the param `rem` removes the restriction. Some important restrictions show the tables 1.3, 1.4, the possible option valuess shows the table 1.5.

Say, the script wants to prevent flying. The restriction is '`fly`', setting is done by param '`add`'. Hence the script sends an onwer message "`@fly=add`". It tells the viewer: Open that special preferences page and check the box for 'No fly'. From now on, the fly button is inactive, the avatar can't fly even in god mode. Until the user loggs out or the script sends the message "`@fly=rem`".

RLV knows also exceptions. They are set very like restrictions, and if set they constrain the effect of restrictions in some way. For example the effect of the restriction '`tplure`' is to refuse automatically TP offers. Well, say we want not to refuse them

[10]The state lasts until either the object is unworn that sat the restriction or the viewer logged out

Restriction	Effect
sendchat	Prevent sending chat messages
chatshout	Prevent shouting
chatnormal	Prevent chatting at normal volume
chatwhisper	Prevent whispering
emote	Exception: Do not truncate emotes
redirchat:<chan>	Redirect public chat to private channels
rediremote:<chan>	Redirect public emotes to private channels
sendchannel[:<chan>]	prevent using chat channel except ...
recvchat	Prevent receiving chat messages
recvchat:<UUID>	Exception: Allow receiving chat from ...
recvemote	Prevent seeing emotes
recvemote:<UUID>	Exception: Allow seeing emotes from ...
sendim	Prevent sending instant messages
sendim:<UUID>	Exception: Allow sending IM to ...
recvim	Prevent receiving instant messages
recvim:<UUID>	Exception: Allow receiving IM from ...
showinv	Prevent using inventory
viewnote	Prevent reading notecards
viewscript	Prevent opening scripts
viewtexture	Prevent opening textures
edit	Prevent editing objects
rez	Prevent rezzing inventory
tplm	Prevent teleporting to a landmark
tploc	Prevent teleporting to a location
tplure	Prevent teleporting by a friend
tplure:<UUID>	Exception: Allow teleporting by ...
accepttp[:<UUID>]	Auto-accept teleport offers from ...
sittp	Limit sit-tp to 1.5m max
sit	Prevent sitting down
unsit	Prevent standing up
fly	Prevent flying
fartouch	Limit touching to 1.5m max

Table 1.3: RLV restrictions and their effect, part 1

Restriction	Effect
showworldmap	Prevent viewing the world map
showminimap	Prevent viewing the mini map
showloc	Prevent knowing the current location
shownames	Prevent seeing the names of the people around
showhovertextall	Prevent seeing all the hovertexts
showhovertext:<UUID>	Prevent seeing one hovertext in particular
showhovertexthud	Prevent seeing the hovertexts on HUD
showhovertextworld	Prevent seeing the hovertexts in-world
detach[:<name>]	Lock the attachment point
addattach[:<name>]	Lock the attachment point empty
remattach[:<name>]	Lock the attachment point full
addoutfit[:<part>]	Prevent wearing clothes
remoutfit[:<part>]	Prevent removing clothes
setdebug	Prevent changing some debug settings
setenv	Prevent changing the environment settings

Table 1.4: RLV restrictions and their effect, part 2

Action or Restriction	Option values
detach, addattach, rematttach	chest, skull, left shoulder, right shoulder, left hand, right hand, left foot, right foot, spine, pelvis, mouth, chin, left ear, right ear, left eyeball, right eyeball, nose, r upper arm, r forearm, l upper arm, l forearm, right hip, r upper leg, r lower leg, left hip, l upper leg, l lower leg, stomach, left pec, right pec, center 2, top right, top,top left, center, bottom left, bottom, bottom right
addoutfit, remoutfit	gloves, jacket, pants, shirt, shoes, skirt, socks, underpants, undershirt, skin, eyes, hair, shape, alpha, tattoo
setdebug*, setenv*	Please, check out RLVa, [9]

Table 1.5: Option values for actions and restrictions

for special persons. The exception for that is ‘`tplure:<UUID>`’, hence after receiving the owned message “`@tplure:eb66b7b7-7ddb-4c5a-95ad-bfeb9837ae29=add`” the viewer not refuses the TP offer coming from Jenna Felton[11].

1.5.5 RLV commands

RLV commands are single-line commands that are handled by the RLV plugin.

1.5.5.1 Restriction management: ADD, REM

```
1 add (, <restr>)+
2 rem (, <restr>)+
```

These two commands allow to add and remove RLV restrictions, tables 1.3, 1.4 and 1.5. For example, to set the restriction `unsit` (making the avatar unable to stand form chars etc.) but remove the restrictions `showinv`, `edit`, `rez` (making the avatar able to see inventory, rez and edit stuff again, this abilities must have been restrictd before), we need this two command lines:

```
1 add unsit
2 rem showinv,edit,rez
```

The TRP interpreter, i.e. the RLV plugin does not parse and check the restrictions for being executable, it just relies them to the viewer. If the viewer can’t execute the command, it shows an error message. This way the TRP interpreter is already ready for commands, that come to the RLVa in future.

1.5.5.2 Action enforcing: FORCE

```
1 force (, <actn>)+
```

This command triggers a RLV action. The command usage is very similar to the usage of restrictions: The command takes the actions to trigger as parameter. To make the avatar stand-up, you need the action `unsit`. Forcing it is done by this commmand line:

```
1 force unsit
```

Similar to the restriction management, the TRP interpreter does not parse the restriction for being correct. This way the interpreter is ready to enforce actions that will come later to the RLVa.

[11] her UUID was used here

1.5.5.3 Clothing management: WEAR, UNWEAR

```
wear (, <name>)+
unwear (, <name>)+
```

These commands work with items in the device inventory. This allows to work with the avatar outfit in a much easier way. Basically, after reading the `wear` command, the interpreter gives the named items to the avatar in usial way, before the viewer is forced to wear (or attach) the items by invoking RLV actions. But details are more complicated.

The RLVa has no actions to attach or wear a single item, it is only possible to wear a folder. Thus, the items must be given in a folder. Giving a huge amount of folders is annoying, hence, the interpreter caches all `wear` commands until any other command is read. Than the inventory of cached `wear` commands is given in a single *container* that is attached as soon it is found in the avatar's inventory.

The `wear` command is blocking: The avatar is given the container each 15 seconds until it was accepted, found in the avatar's inventory and attached. Only after that the program execution continues.

The `unwear` command works differently. Nothing is cashed, the command does not block and just triggers the RLV actions detaching or undressing the named inventory.

Both commands work with any type of wearable inventory in same way, you can mix body parts (shape, skin, eyes or hair), any layered clothing and any objects that are handled as attachments. Since body parts can't be unworn, you can not use them in the `unwear` command.

Let's give a short example. Say the device is made for transforming the avatar into a marble statue. For that the avatar must wear a skin, eyes, hair, a jug and a statue base. This items must be available in the device inventory. Wearing them is done by this commands:

```
wear White Marble Skin, White Marble Eyes, White Marble Hair
wear White Marble Jug (l forearm)
wear White Marble Base Low (left foot)
```

As soon another command is found than `wear` after the line 3, all the named inventory is checked to satisfy requirements, than given into a container folder in the avatars inventory and after the container folder was found, the viewer is forced to wear and attach items into it.

Now, if we want remove the items, we should write this command:

```
unwear White Marble Jug (l forearm)
unwear White Marble Base Low (left foot)
```

If the commands are found, the RLV plugin finds out where the items are attached and tells the viewer to detach or undress the items from that attachment points of clothing layers.

Requirements

The `wear` and `unwear` commands sets requirements to the inventory items:

1. Every named item must be in the device inventory and the owner must have a *copy* permission on it. SL refuses giving no-copy items in a folder.
2. For attachments, the item name must specify a valide name of the attachment points in round brakets, for example "Skirt (Pelvis)". Otherwise the interpreter fails to resolve attachment point to unwear.
3. For clothes, the item name must contain the valide name of clothing layer, also in round brakets, for example "Latex strings (underpants)". Otherwise the interpreter fails to resolve clothing layer to undress.

Names of attachment points and clothing layers you'll find in the table 1.6.

1.5.5.4 Clothing removal: STRIP

```
1 strip (, <part>)+
```

This command allows to undress clothes, attachmetns and groups of them in similar way, simplifies so the actions. The command expects names of individual clothes layer, attachment points or groups. For example to remove all objects, attached to the HUD, we can just use this command line:

```
1 strip huds
```

If the interpreter reads it, it produces a large number of removing actions cleaning each HUD attachment pont. The table 1.6 displays names of indivdual items and item groups you can use as command parameter.

1.5.5.5 Teleport command: TP2LM

```
1 tp2lm <name>
```

This command takes name of a landmark in the devise inventory and forces a teleport to it's target place. If the place is secured by a landing point, the agent is also warped and brought quite close to the landmarked position. Example:

```
1 tp2lm Dreamland of Protection
```

Group	Clothing layer or attachment point
clothes	gloves, jacket, pants, shirt, shoes, skirt, socks, underpants, undershirt, alpha, tattoo
huds	center 2, top right, top, top left, center, bottom left, bottom, bottom right
head	skull, mouth, chin, left ear, right ear, left eyeball, right eyeball, nose
body	chest, left shoulder, right shoulder, left hand, r upper arm, r forearm, l upper arm, l forearm, right hip, r upper leg, r lower leg, left hip, l upper leg, l lower leg, stomach, left pec, right pec, right hand, left foot, right foot, spine, pelvis
objects	huds, head, body I.e. every individual item in this groups

Table 1.6: Group names and individual items in each grop

In this case the landmark name is “*Dreamland of Protection*”. The commmand is blocking: If the named lanmark is missing, the TP attempt is not started, otherwise the executon continues only if the aggent arrives at landmarked place or was brought as close to it as posible.

Teleport note

Teleports in SL are adventurous at times, please use this command wiselly: Sometimes the target sim was not fast enough to accept the avatar, we notice it by manual teleports if the error message pops up. The interpreter can’t receive this message and will keep teleporting until success, but logging out the avatar is than also possible. Lindens work on this problem but if the grid or the target sim is overload, the failure might hapen.

Warp notes

If the teleport succeeds, the task is not always completed: Some locations may use landing points that catch the teleported in agents. It is good for the malls or clubs but not for roleplays. Thus, the interpreter trys to warp the agent to targetted position. Warping means, the avatar is pushed towards the place with a speed, high enough to break through solide walls[12].

The warping process is aborted in three cases:

[12] A speed of about 150 meters per second was measured

1. The warp route crosses a wall or other object that is too solide and the avatar was not able to bring through, the interpreter gives up and leaves the avatar beofore the wall
2. The region allows damage or the warp route crosses a parcell with allowed damage. Touching solide prims while warping would kill the avatar and bring them home, thus the RP continues at the landing point or reached parcell with switched on damage
3. The avatar is flying. Pushing does not work than and the avatar is left at landing point

While awoiding the first two reasons needs good planing the RP in a way the warping route is free from solide objects or damaging parcells, you can work around the third reason by restricting the fly ability before teleport and releasing it after the teleport is completed. The example program will become this:

```
1 add fly
2 tp2lm Dreamland of Protection
3 rem fly
```

1.5.6 RLV conditions

If the RLV plugin is installed, you get access to two more condition operatorss.

1.5.6.1 RLV check: RLV

```
1 (if | while) [not] rlv
2 <constraint>
```

The condition `rlv` takes no parameters. It resolves to number 0 (true) if the user runs a RLV compatible viewer and it resolves to number 1 (false) if the RLV-compativle viewer was not found, example:

```
1 if rlv
2     wear Marble-skin, Marble-eyes, marble-hair
3     ...
4 else
5     message no RLV found, the transformation will not work!
6 default
7     message no response from RLV plugin.
8     think Possible script crash. Closing RLV support
9 end
```

Please always check for RLV if you want use RLV operations, at least the blocking `wear` and `tp2lm` commands, to prevent hanging up the program.

1.5.6.2 Clothing check: WORN

```
(if | while) [not] worn <part>
<constraint>
```

The condition `worn` takes a name of clothing layer, attachment point or a group of them and resolves to 0 (true) if the point or layer or any element of the group is used. The parameter `part` is quite thesame as used as param of `strip` command, table 1.6.

How the condition works:

- If the parameter `part` takes a name of any special clothes layer or attachment poing, than the condition resolves to 0 (true) if this layer or attachment point is used and to 1 (false) if the part is empty.
- If the parameter `part` takes a name of a group of clothes layers or attachment points, than the condition resolves to 0 if any part in the group is used by a clothing or attachment and it resolves to 1 if every part in the group is empty.

Working example:

```
if worn clothes
    whisper Glad you wear something, nothing to do
else
    whisper Hey, you are NUDE! let's change this...
    wear Jeans (pants), Shirt (shirt), Sneakers (shoes)
default
    message No idea if you wear something, we hope you do
end
```

1.6 Script API

This section is for developers who know how to make scripts using linkd messages. Every script interacting with the TRP interpreter this way is assumed to be a (device) driver, if it controls any device function or not. There are two categories of messages the TRP interpreter supports: *control* and *interactive* messages.

1.6.1 Control messages

Control messages are initiated by the drivers or the interpreter and sent via linked messages with this properties:

- The Interpreter uses `LINK_THIS` as message target.
- The number parameter is -2018160314 (`TRPCN` in numbers).

1.6.1.1 Hard reset: RESET

Ingoing[13], message: '`RESET`'. The effect of this message is hard-resetting of TRP scripts. Example LSL command to be executed by the driver:

```
llMessageLinked(LINK_THIS, -2018160314, "RESET", NULL_KEY);
```

1.6.1.2 Enforced stop: STOP

Bidirectional[14], message text: '`STOP`'. Stopps the execution immediatelly and releases all RLV restrictions. Example call:

```
llMessageLinked(LINK_THIS, -2018160314, "STOP". NULL_KEY);
```

1.6.1.3 Enforced start: RUN

Bidirectional, message: '`RUN`', message key: Name of the program to start. An example message starting start a program '`strawberry`':

```
llMessageLinked(LINK_THIS, -2018160314, "RUN". "strawberry");
```

The program name is case-sensitive, extended by '`.trp`' gives the notecard name to execute, in this case '`strawberry.trp`'.

[13]The interpreter can only receive this message
[14]The interpreter can send and receive this message

The notecard is started if found and is executable. Otherwise the interpreter sends a `STOP` command. Like for the `start` TRP command, a notecard is executable if it has content and is full-perm in device inventory.

1.6.2 Interactive messages

Interactive messages are initiated by the program and sent via linked messages with this properties:

- The message target is given by the TRP commands.
- The number parameter is every number but -2018160314.

1.6.2.1 Command messages

A command message is *non-blocking*, sent due executing the `lm` TRP command, section 1.5.2.4. Look there please for definitions and example.

```
1 lm <link> <number> <text>
```

Executing this TRP command results in executing this LSL command:

```
llMessageLinked(<link>, <number>, "<text>", NULL_KEY);
```

1.6.2.2 Request messages

A request message is *blocking*, sent due resolving the `lm` condition, section 1.5.3.6. For definitions and usage example, look there please.

```
1 (if | while) lm <time> <link> <number> <text>
```

Resolving this condition starts the timer and sends linked messabe via:

```
llSetTimerEvent(<time>);
llMessageLinked(<link>, <number>, "<text>", NULL_KEY);
```

Now two things can hapen. Either the timer runs out first, than the condition is unresolved, or a response arrives via linked message send by executing this LSL command by device driver:

```
llMessageLinked(LINK_THIS, <number>, (string)"<resp>", "<key>");
```

The value of `resp` must be a number, the condition resolves to: 0, if it resoves to `true`, 1 if it resolves to `false`, a number larger 1 for further cases, or give an empty string `""` if the condition could not be resolved. The value of `key` is any string but `NULL_KEY`, e.g. an empty string.

1.7 Example driver

At this place we look at the code of the driver script 'waterglass.drv'. As the code is too long for a page, we do it peacewise.

1.7.1 Preamble

```
1  //---------------------------------------------------------
2  // $RCSfile: waterglass.drv.lsl,v $
3  //
4  // TRP device driver: Glass of mineral water.
5  //
6  // ->[]    <-2018160314, "RUN", name>
7  // ->[]    <-2018160314, "STOP", NULL_LEY>
8  // ->[]    <0, "drink", NULL_LEY>
9  // ->[]    <0, "isempty?", NULL_KEY>
10 //    []-> <0, "0", "isempty?">
11 //    []-> <0, "1", "isempty?">
12 // ->[]    <0, "detachme", NULL_LEY>
13 // ->[]    <0, "dropme", NULL_LEY>
14 // ->[]    <0, "placeme", NULL_LEY>
15 //
16 //---------------------------------------------------------
17 // Author  Jenna Felton
18 // Version 1.0, $Revision: 1.3 $
19 //         $Date: 2010/09/30 18:09:50 $
20 //---------------------------------------------------------
```

Figure 1.4: Device driver preamble

The first part is the preamble, figure 1.4. Here is the actual script API specified. As pure comment code it has no effect as a script, but writing this part is usefull for distributed development and for remembering what the script does after weeks doing something else. Let's introduce the syntax choosen by the author years ago.

The acute brackets '<...>' mean a linked message. Messages the skript receives are noted after the prefix '->[]', messages the script sends after the prefix '[]->'. The three components inside the brackets denote the number, string and key parameter of the linked message. The target parameter is omitted. For example the message '<0, "1", NULL_KEY>' is sent by executing the LSL command

```
llMessageLinked(LINK_THIS, 0, "1", NULL_KEY);
```

The rest is a bit CVS, common informations and good place to mentoin changes.

1.7.2 Animation management

1.7.2.1 Why outsourcing animations?

The TRP language has no command for animating the avatar or triggering sounds or particle effect etc. There are two reasons for it: First, this makes the language smaller. Second, things like animations or sound are special for the given device. There is often more to do, than just making the avatar drink. Drinking means also lovering the volume of water into the glass, i.e. animating the avatar comes also with changing the device state.

This is the main reason why management of animations (and sounds and other similar things) is the job of device drivers. The interpreter triggers the tasks by sending interactive messages, section 1.6.2.

1.7.2.2 Back to the code

The second piece of code manages the volume of water and triggers drinking animation, figute 1.5.

The water in the glass is visualized by a prim, the prim is formed in a way, it must just lower the prim high to display decreasing water. The actual prim high saves the variable `fWaterZDim`.

Now, the function `getDrink()` animates the avatar, decreases the value of the variable and actualizes the high of the water prim, lowering so the visual volume of content into glass.

The function `isEmpty()` is called to resolve if the glass is empty or has stil some water. The function simply checks if the value of `fWaterZDim` variable is greather 0 or not. The answer is sent directly via linked message, the TRP interpreter understands.

Both functions offer a service to the the example program, used by sending interactive messages. The code is at lines 14 to 17, figure 1.3:

The command "`while not lm 15 this 0 isempty?`" at line 14 produces a request message: The interpreter sends a command '`isempty?`' via linked message and stops. This linked message is received in the default state of the driver code, figure 1.7, and results in calling the function `isEmpty()`. The function sends the answer via linked message, the interpreter receives the answer and decides if to repeat the loop or leave.

The command "`lm this 0 drink`" at line 15 produces a command message: The interpreter sends a command '`drink`' also via linked message, not stops here but in the line 14 (the `wait` command is another story). However, the sent message is received

also by the driver, and the function `getDrink()` is called. While the interpreter waits, the avatar runs drinking animation and the amount of water decreases.

The whole interaction repeats until the glass is empty and the loop left.

1.7.3 Detachment management

1.7.3.1 Why outsourcing detachment actions?

Sometimes, if you equip an attachment by TRP, you want run some code arter detaching it. The simplest example is this TRP program segment (detaching via interpreter only requires using RLV):

```
1 # detaching via RLV
2 force detachme
3
4 # terminating
5 clean script
```

Why the code is not good? Here, only one command will successfully run, either the detaching command or the terminating command with fixing of scripts. There is no possibility to have both completed.

A better idea is let the driver detach the device. The RLV action is just replaced by sending a comand message and he program segment looks like this:

```
1 # detaching via driver
2 lm this 0 detachme
3
4 # terminating
5 clean script
```

The driver would receive the message and detach the glass than. But attention. If the driver does that immediatelly, than we changed nothing on the dilemma above. Instead, the driver should notice the detachment request and perform it as soon the interpreter is ready with terminating code: While the interpreter executes the `clean` command, it sends the `STOP` message too. The driver must wait for this message and detach the device than.

1.7.3.2 Back to the code

The third piece of code manages detachment requests of the interpreter, figute 1.6.

The code implements the solution, discussed above, but has to handle a few more possibilities: Due of lag between scripts, the detachment request could reach the

driver before or also after the STOP message. Also there are three different detachment requests, two of them run additional actions. The program code calling them is at lines 21 to 33, figure 1.3.

The command "lm this 0 detachme" at line 22 sends a command message 'detachme'. Similarly, the commands at lines 25, 28 and 30 send command messages 'dropme', 'placeme' and also 'detachme'. The termination command "clean script" at line 33 sends the command message 'STOP'.

Each message is received by the driver and results into the call of the function detachMe() with values of DA_DETACH, DA_DROP, DA_PLACE and DA_STOP as parameter respectivelly. The function does not perform the detachment request immediatelly but buffers the requests and waits until it receives both, any detachment request and the stop request. In case of special detachment request, it performs the additional actions, dropping and placing the glass.

How that actions work? Actually, scripts can only detach an attachment into inventory and never strip it to the floor, thus to emulate this actions, the driver script rezzes a dummy object, and detaches the glass. The dummy glass looks quite like an empty glass and completes the requested operation.

1.7.4 Default state

The last piece is the default state, putting all pieces together, figute 1.7. What hapens here.

The first thing the state do is requesting the owner permissions. Since it is an attachment, the script is granted them automatically: The permission to detach the device and the permission to animate the avatar. If the owner changes the script resets to request permissions again.

The animation 'hold_R_handgun' is the endless holding glass animation, it is started immediatelly if the glass is attached and stoped if the glass was detached.

Every requested action is hapens if the driver is requested via linked message, each action is handled in the prevously introduced functions, hence the link_message event does nothing but delegates the commands to the right place.

1.7.5 Dummy glass

Well, actually the driver code is introduced, but a small detail is mising stil: The dummy object to rezz. The item looks exactly as the regular glass but without the water prim (only the empty glass is droped). The glass will be physical in order it falls down nicelly and it should be temporary in order the parcell is not overfilled with used dishes. This item is inside the glass in order the driver can rezz it.

We can actually let the item unscripted, it will look good if the avatar wants trow the glass, but if the it was 'placed', it should take the standing position but because of being physical, it could tilt over. Not nice, better to make it non-physical and bring into a standing position as soon it collides with table surface. This will manage a short script inside the helping glass, figure 1.8.

Whilie initialization, the script sets the glass automatically to temporary and physical (so we not forget it). While rezzing the item is told, if it was thrown or placed, the script notices that. Because of being physical, the glass will fall down regularly and as soon it touches the table or floor surface, the script decides, if to living it roll around or make non-physical and standing.

However, if you are equiping the supplied example glass with TRP as described in the section 1.3.2, you don't need making this helping glass or placing this script into, its done already.

Thats all

Thank you for reading the manual. All success with equiping your products with TRP wishes you your JFS shop.

```
22 integer WATER_PRIM      = 2;
23 float   WATER_DIAM      = 0.049;
24 float   DRINK_STEP      = 0.015;
25 float   fWaterZDim      = 0.2;
26
27 getDrink() {
28     if (fWaterZDim > 0.0) {
29         llStartAnimation("drink");
30
31         llSleep(1.0);
32
33         fWaterZDim -= DRINK_STEP;
34
35         if (fWaterZDim > 0.0) {
36             llSetLinkPrimitiveParams(WATER_PRIM,
37                 [PRIM_SIZE, <WATER_DIAM, WATER_DIAM, fWaterZDim>]);
38         }
39         else {
40             llSetLinkAlpha(WATER_PRIM, 0.0, ALL_SIDES);
41         }
42     }
43 }
44
45 isEmpty() {
46     if (fWaterZDim > 0.0) {
47         llMessageLinked(LINK_THIS, 0, "1", "isempty?");
48     }
49     else {
50         llMessageLinked(LINK_THIS, 0, "0", "isempty?");
51     }
52 }
```

Figure 1.5: Animation management

```
54 vector  REZ_OFFSET      = <0.3, -0.3, 0.5>;
55 vector  REZ_VELOCITY    = <0.25, 0.0, 0.0>;
56
57 integer DA_DETACH       = 1;
58 integer DA_DROP         = 2;
59 integer DA_PLACE        = 3;
60 integer DA_STOP         = 4;
61 integer gDetachActn     = FALSE;
62
63 detachMe(integer actn) {
64     if (!gDetachActn) gDetachActn = actn;
65     else {
66         if (actn == DA_STOP) {
67             if (gDetachActn == DA_STOP) return;
68             else                        actn = gDetachActn;
69         }
70         else if (gDetachActn != DA_STOP) return;
71
72         if (actn == DA_DROP || actn == DA_PLACE) {
73             string inv = llGetInventoryName(INVENTORY_OBJECT, 0);
74             if (inv != "") {
75                 rotation rot = llGetRot();
76                 vector   pos = llGetPos();
77                 vector   vel = ZERO_VECTOR;
78
79                 pos = pos + REZ_OFFSET*rot;
80                 if (actn == DA_DROP) vel = REZ_VELOCITY*rot;
81
82                 llRezAtRoot(inv, pos, vel, ZERO_ROTATION, actn);
83
84                 if (llGetInventoryType(inv) == INVENTORY_OBJECT) {
85                     llRemoveInventory(inv);
86                 }
87             }
88         }
89
90         llDetachFromAvatar();
91     }
92 }
```

Figure 1.6: Detachment management

```
94  default {
95      state_entry() {
96          llRequestPermissions(llGetOwner(),
97              PERMISSION_ATTACH|PERMISSION_TRIGGER_ANIMATION);
98
99          llSetLinkAlpha(WATER_PRIM, 0.5, ALL_SIDES);
100         llSetLinkPrimitiveParams(WATER_PRIM,
101             [PRIM_SIZE, <WATER_DIAM, WATER_DIAM, fWaterZDim>]);
102     }
103
104     changed(integer change) {
105         if (change & CHANGED_OWNER) llResetScript();
106     }
107
108     run_time_permissions(integer perm) {
109         if(perm|PERMISSION_TRIGGER_ANIMATION) {
110             llStartAnimation("hold_R_handgun");
111         }
112     }
113
114     attach(key id) {
115         if (id == NULL_KEY)    llStopAnimation("hold_R_handgun");
116         else                   llStartAnimation("hold_R_handgun");
117     }
118
119     link_message(integer sender, integer num, string msg, key id) {
120         if (num == 0) {
121             if      (msg == "isempty?")  isEmpty();
122             else if (msg == "drink")     getDrink();
123
124             else if (msg == "detachme")  detachMe(DA_DETACH);
125             else if (msg == "dropme")    detachMe(DA_DROP);
126             else if (msg == "placeme")   detachMe(DA_PLACE);
127         }
128
129         if (num == -2018160314) {
130             if (msg == "RUN")            llResetScript();
131             if (msg == "STOP")           detachMe(DA_STOP);
132         }
133     }
134 }
```

Figure 1.7: The default state

```
//-----------------------------------------------------------
// $RCSfile: dummyglass.lsl,v $
//
// Dummy glass for mineral water.
//
//-----------------------------------------------------------
// Author  Jenna Felton
// Version 1.0, $Revision: 1.1 $
//         $Date: 2010/09/30 13:27:47 $
//-----------------------------------------------------------

integer DA_PLACE        = 3;
integer placed          = FALSE;
key     owner;

default {
    state_entry() {
        llSetPrimitiveParams([
            PRIM_PHYSICS, TRUE,
            PRIM_TEMP_ON_REZ, TRUE]);
    }

    on_rez(integer param) {
        if (param == DA_PLACE) placed = TRUE;
        owner = llGetOwner();
    }

    collision_start(integer num) {
        if (!placed) return;
        while (num-- > 0) {
            if (llDetectedKey(num) == owner) return;
        }
        llSetPrimitiveParams([
            PRIM_PHYSICS, FALSE,
            PRIM_ROTATION, ZERO_ROTATION]);
    }
}
```

Figure 1.8: Script inside dummy object

2 Deutsch

2.1 Einleitung

The Red Pill[1], im folgenden nur *TRP*, ist ein Projekt, der mit einer Idee anfing, Geräte zu entwickeln die den Avatar zu umwandeln vermögen. Der Projektname kommt von der roten Pille die Neo im Film Matrix genommen hat. Diese Pille änderte schlagartig Neo's Leben. Etwas ännliches müsste die rote Pille in SL auch tun.

Das soll eifach ein Attachment sein, das den Avatar umwandelt, ob in eine Katze, einen Vogel, Roboter, eine Wolke oder Feuerball. Schritt für Schritt, mit Nachrichten zwischendurch und Ausführung von Animationen. Innerhalb von Minuten oder auch Stunden. Automatisch und immersiv. Wichtiger war es aber, nicht einfach ein Gerät für eine bestimmte Transformation zu entwickeln, sonden eine Technologie zum Entwickeln derartigen Geräte.

Um diese Ziele zu erreichen wurde die Programmierung der Umwandlung von der Umwandlung selbst getrennt: Der Vorgang wird in einer Notekarte als Befehlsfolge beschrieben. Diese wird durch einen Skript ausgelesen und ausgeführt. Die Befehle sind einfach zu erlernen, neue Transformationsszenarien können schneller entworfen oder bestehende angepasst werden.

TRP ist demnach ein Interpreter der die Programnote ausliest und die Befehle darin auszuführt. Dadurch eignet er sich auch gut zum Entwickeln von Geräten die zeitlich gesteuerte Aktionen ausführen wie Öffnen von Türen, Versenden von Chatnachrichten, Schalten von Licht und ännliches. In diesem Dokument wird das Entwickeln eines Getrenk vorgestellt, wobei das austrinken des Inhalts und ablegen des Getrenkglass durch TRP gesteuert wird.

Kurzer Befehlsüberblick

- Chatbeefehle: Rufen, Sagen, Flüstern, privates Chat
- Flußkontrolle: `if`, `for`, `while`
- Benutzerinteraktion via Dialoge
- Gerätekontrolle und -interaktion via Linknachrichten

[1] Die Rote Pille

- Setzen, entfernen von RLV Restrictions und erzwingen von RLV Aktionen
- Ändern des Körpers und Kleidung des Avatars
- Prüfung auf Benutzung von RLV und Tragen von Kleidung oder Attachments

2.1.1 Paketinhalt

Der Inhalt der Produktbox ist in 1.2 abgebildet.

- Skripte `trp.core`, `trp.rlv` und `trp.emergence`
- Lizenzkarten `JFS Developer License Agreement`, `JFS Entwickler Lizenzvereinbarung`
- Hilfsnote `TRP SE Help` mit eingebetteten Notekarten in englisch und deutsch: Benutzerhandbuch, Tutorial, Spezifikation der TRP Sprache und API
- Tutorialobjekte: Objekt '`waterglass`', Note '`autorun.trp`' und Skript '`waterglass.drv`'
- Landmarke zum JFS main store
- Skript '`*productKey`' für Produktupdates und Keyholder, bereits mit diesem Skript geladen.

2.1.1.1 Scriptrechte (Entwicklerversion)

Die Skripte '`trp.core`' und '`trp.rlv`' werden mit Benutzerrechten *copy, no-mod, transfer* vertrieben, Abbildung 2.2. Bei dieser Einstellung laufen sie nicht. Um sie zu 'aktivieren', müssen die Rechte des Nächstbeutzers der Skripte auf entweder *no-copy* oder *no-transfer* reduziert werden. Das hindert Sie nicht, eigene Entwicklungen zu vertreiben die damit ausgerüdtet sind, und auch nicht, beliebige Inworldrechte des Nächstbenutzers diesen Entwicklungen zu setzen.

2.1.2 TRP - Architectur

Die TRP Architektur ust unter 2.1 abgebildet.

Die Architektur ist aus vier Schichten aufgebaut. Die obere Schicht ist das *TRP Program*, enthaltend in einer Notekarte. Jeder Programname ist möglich, nicht nur '`program`', allerdings muss der Name der zugehörigen Notekarte das Programname sein, erweitert mit '`.trp`'. Der Name '`autorun.trp`' ist besonders: Eine Notekarte mit diesem Namen wird automatisch gestartet.

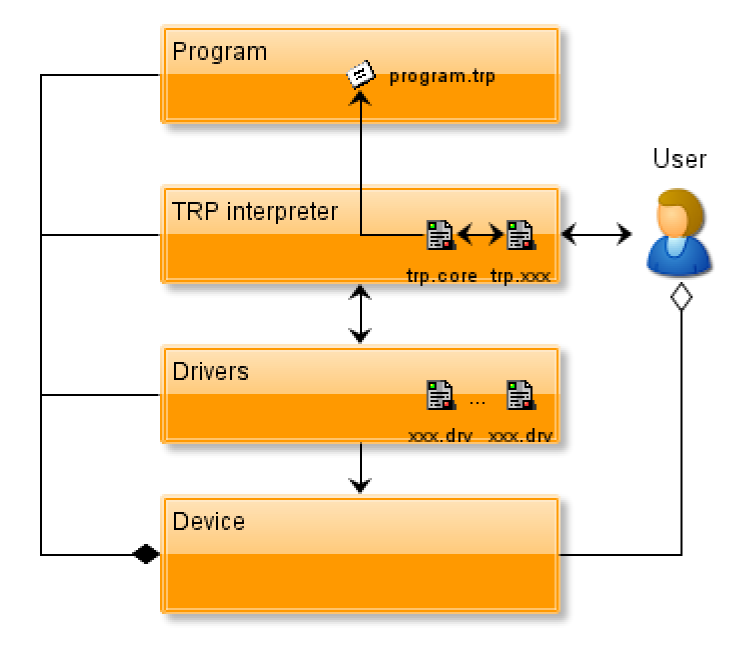


Figure 2.1: TRP - Architectur

Die zweite Schicht ist der *TRP interpreter*, bestehend aus dem Interpreterskript (bzw. *Kernscript*) '`trp.core`' und verschiedenen Pluginskripten. Ein Plugin ist ein geräteunabhängiger Skript der für Ausführung bestimmter Programmbefehle oder zum Unterstützen des Interpreters und anderer Plugins zuständig ist. Pluginskripte sollten den Namenpräfix '`trp.`' verwenden.

Die dritte Schicht bilden die *Gerätetreiber*. Das sind geräteabhängige Skripte, die für die Interaktion zwischen dem Interpräter und dem Gerät zuständig sind. Sie sind kein Teil des TRP Interpreters, sondern der TRP Installation. Treiberskripte müssen den Namenssuffix '`.drv`' verwenden.

Die untere Schicht ist der *Systemträger*, das ist das Gerät selbst. Es gehört dem Benutzer und beinhaltet physikalisch die Programnote, den Interpreter und die Treiber.

2.1.3 Immersion und RLV

Die Avatartransformation ist einfach ein Wechsel der Körperteile wie des Shape, der Skin, Attachments und AO. Geschieht das aber manuell, lässt dies wenig Raum für die immersive Erfahrung. Wird außerdem der reguläre Viewer verwendet, besitzt der Benutzer zu jedem Zeitpunkt die volle Kontrolle über den Prozess. Das macht jedes Gefühl zunichte, transformiert zu werden.

Glücklicherweise existieren so genannte RLV-kompatible Viewer in SL, Abschnitt 2.1.3.3. Eigentlich ist *RLV* der Name des ersten Viewers, der den regulären SL Viewer um eine 'erwachsene' Funktionalitt erweiterte. Diese Funktionalität wurde als eine API[2] zusammengefasst und veröffentlicht und nun auch von anderen Viewern implementiert, [9]. Aus diesem Grund werden Viewer, die dieses API implementieren, RLV-kompatibel bzw. einfach *RLV* bezeichnet. Die API selbst wird meist mit *RLVa* bezeichnet, so auch hier.

Warum ist dieses API wichtig für TRP? Allgemein ist ein API ein Interface zwischen Maschinen oder Programmen. Ein Program wie der Viewer bietet ein Kontrollinterface für Programme wie Skripte. Das erlaubt es dem TRP Interpreter, auf bestimmte Funtionen des Viewers zu zugreifen, um das immersive Gefühl der Transformation zu vermitteln.

Es ist nicht erforderlich, für TRP-Geräte den RLV-kompatiblen Viewer zu nutzen. Tatsächlich wurde der Code, der auf RLVa zugreift, in den Skript '`trp.rlv`' untergebracht, den *RLV plugin*. Ist dieser Skript nicht installiert, funktionieren die TRP-Geräte dennoch, aber ohne Zugriffe auf RLVa und daher den Viewer.

2.1.3.1 Das kann RLVa (einige wenige Beispiele)

- Sachen wie Shape, Skin, Attachments oder Kleidung in einen ausgezeichneten Inventarordner des Avatar annehmen
- Automatisches Anziehen von Sachen in diesem Ordner
- Verhindert das wechseln von Kleidung oder Attachments
- Automatisches ablegen von Attachments und Kleidung
- Verhindert das Fliegen, Berühren entfernter Objekte
- Verhindert Zugriffe aufs Inventar, Editieren oder Rezzen von Objekte
- Blokiert IM und Chat Unterhaltung
- Verbirgt Karten, anzeige der Avatarposition und der Avatarnamen
- Schränkt ein oder erzwinkt die Umgebungseinstellungen
- Automatisches gridweite Teleport wie durch benutzung einer Landmarke

[2] Application programing interface

2.1.3.2 Das kann RLVa zum Beispiel *nicht*

- Sachen im Inventar zerstören oder weitergeben
- Geld unerlaubt ausgeben
- Private IM-Konversationen preisgeben
- Passwörter stehlen oder Kreditkartendaten[3]

2.1.3.3 RLV-kompatible Viewer

Es gibt sicher mehr, aber diese sind verbreitet, dem Author[4] bekannt und bekannt, RLVa implementiert zu haben:

- Klassisches RLV, [8]
- Imprudence, [3]
- Cool Viewer, [1]
- Rainbow Viewer, [10]
- Phönix viewer, [7]
- Emerald Viewer, [2]

Diese Viewer sind von Drittanbieter, die LL vertreibt sie nicht. Um die Entwickler zum Entwickeln vertrauenswürdigen Viewer zu ermutigen, hat LL die Police der Drittanbieterviewer herausgebracht (*TPVP*, [12]), so wie das Verzeichnis von Drittanbieterviewer (*TPVD*, [11]). Das TPVD enthält die Viewer, deren Entwickler eine Selbstverpflichtung zur Einhaltung des TPVP eingegangen sind.

Die beiden ersten Viewer, den RLV und Imprudence, stehen auf der TPVD, das bedeutet, ihre Entwickler versichern, dass die Viewer vertrauenswürdig sind.

Der Cool Viewer und Rainbow Viewer waren nie auf der Liste, die Gründe fnden sich auf der Webseiten der Entwickler. Beide Viewer waren einst derselbe, dann trennte sich die Entwicklung, die Geschichten darüber finden sich sicher auch dort. Der Cool Viewer wird immer noch weiterentwickelt, der Rainbow Viewer nicht mehr.

Der Emerald Viewer war mal auf der Liste, dann wurde daraus genommen, nachdem einiges ereignet hat, das mit dem Policy nicht vereinbar war, und hat auch das Recht verloren, auf den SL Grid zugreifen zu können . Er wird auch nicht mehr weiterentwickelt. Der Viewer war eine hervorragende Entwicklung, er war sehr weit verbreitet, und es wird bestimmt auch Abkömmlöinge geben, einer ist zum Beispiel der Phönix Viewer.

[3]Zumindest verbreitete Viewer solch dumme Sachen nicht tun
[4]Jenna Felton

2.1.4 Skriptverbotene Bereiche

Vielleicht kennen Sie diese Bereiche bereits, wo alle Geräte ihren Dienst einstellen, außer Fahrzeuge und einigen AOs, weil dort die Ausführung fremder Skripte untersagt ist. Nun, bei vielen Sachen ist diese Einstellung nur lästig, kann sie bei einer Sitzung, wo RLV zum Einsatz kommt, zu auswegslosen Situationen führen (aus der Sicht des Avatars gesprochen).

Solche Situationen sind jene, aus denen der Avatar ohne fremde Hilfe nicht entkommen kann. Bevor man zu diskutieren beginnt, ob solche Situationen möglich oder überhaupt erlaubt seien, stellen Sie sich die Verwandlung einer Raupe in einen Schmetterling vor.

Die Raupe kann nichts anderes tun, außer auf die Verwandlung zu warten. Soll ein Avatar diese Transformation durchmachen, darf er weder fliegen, noch teleportieren, und auch keine Möglichkeit haben, diese Einschränkungen zu umgehen, damit der Spieler sich tatsächlich wie Raupe fühlt.

Nach sagen wir 10 Minuten erscheinen die Flügel und alle Einschränkungen sind aufgehoben. Normalerweise. Nun gerät der Avatar, ob absichtlich oder zufällig, inmitten der Transformation auf einen Ort wo Skripte deaktiviert sind. Aus den 10 Minuten wird eine endlose Ewigkeit.

Damit genau das nicht passiert, dient der *Emergenceplugin*, der Skript '`trp.rlv`'. Er benutzt einen Trick und benimmt sich wie ein Fahrzeug: Er übernimmt die Steuerung des Avatars. Fahrzeugskripte genießen eine Ausnahme und werden auch in den verbotenen Bereichen ausgeführt. Einige AOs funktionieren deshalb auch, sie übernehmen ebenfals die Steuerung.

Allerdings sollte man sehr sparsam mit diesem Plugin umgehen und diesen nur wenn unbedingt erforderlich instalieren. Es gibt eine Reihe guter Gründe, warum der Plugincode nicht in den Interpreterskript aufgenommen wurde.

Der Allgemeine Grund: An vielen Gebieten sind Skripte nicht umsonst verboten. Außer die Besucher zu ärgern kann es einfach sein dass der Server insgesamt beschränkte Resourcen hat und durchs Abschalten von Sktipte mehr Resourcen für Fahrzeuge oder Avatarbewegung freigegeben wird. Außerdem ist man lediglich Gast an diesen Orten und man sollte als Gast den Recht und Gründe des Gastgebers respektieren, Skripte zu verbieten.

Technischer Grund: Der Skript verbraucht ebenfalls Resourcen für die Ausführung. Zum Einen ist da ein aktiver Timer, der regelmäßig prüft, ob die Steuerung noch vorhanden ist und wenn nicht, diese erneut übernimmt. Zum Anderen bedeutet die Übernahme der Steuerung, dass beim Drucken von etwa Pfeiltasten etwas mehr Code ausgeführt wird. Kurz: Verwendung des Plugins trägt ein Wenig zum Lag bei.

Allerdings ist der Skript mit Bedacht geschrieben, möglichst lag-arm zu sein, die Pfüfung auf Steuerung findet alle 15 Sekunden statt, also relativ selten, und es wird nur die Bild-runter Taste übernommen, in der Hoffnung, dass der Server weniger Code ausführen wird da diese Taste recht selten banutzt wird. Und der Skript wird erst aktiv wenn ein TRP Program ausgeführt wird, davor und danach bleibt er nur wartend.

Übrigens, wenn einen Standartviewer benutzt wird, erscheint der button 'Tasten freigeben' am Bildschirm sobald der Skript aktiv wird. Dieser button wird alle 15 Sekunden wieder erscheinen, wenn man ihn wegklickt. Sehr lästig, besonders wenn kein Gerät verwendet wird, das die Steuerung tatsächlich brauchen würde, wie ein Fahrzeug etwa. Erst der RLV und kompatible Viewer haben gelernt, diesen Button auszublenden, soll heißen, wird der RLV Plugin nicht verwendet, braucht es den Emergenceplugin auch nicht.

Und schließlich kann man den Plugin ebenfalls weglassen, wenn im selben Objekt Skripte installiert sind, welche die Steuerung auch übernehmen, wie Fahrzeige oder AOs. Der Plugin würde dann nur unnütz Resourcen verbrauchen.

2.1.5 Versionsinfo

v1.0 (Oktober 2010)

- Initiale Herausgabe

2.2 JFS Entwickler Lizenzvereinbarung

DLA,
version 1.2

Beim Erwerb dieses Produkts akzeptieren Sie folgende Lizezbedingungen:

Gegenstand der Vereinbarung

1.1. Die Lizenzvereinbarung regelt den Vertrieb von JFS Entwicklerprodukte ("*Produkte*") so wie der damit enwickelten Produkte ("*Entwicklungen*").

1.2. "Vertrieb" bezeichnet eine Weitergabe des Produkts oder seiner Bestandsteile in jeglicher Form, unabhängig vom Inhalt das mitvertrieben wird.

1.3. Beim Vertrieb des Produkts muss die Produktquelle angegeben werden, und sollte sein Inhalt verändert werden, auch die vorgenommene Änderungen. Diese Bedingung entfällt wenn die Erfülung den Weitergabeprozess behindert.

1.4. Beim Vertrieb des Produkts muss diesem eine unveränderte Kopie dieser Lizenzvereinbarung beiliegen. Diese Bedingung entfällt wenn die Erfülung den Weitergabeprozess behindert.

Full-perm Produktinhalte

2.1. Produktinhalte, die fullperm vorliegen, müssen nur als fullperm vertrieben werden, unabhängig vom Inhaltstyp und ob sie getrennt vertrieben wird oder mit anderen Entwicklungen oder in diesen.

2.2. Produktinhalte entsprechend der Bedingung 2.1. dürfen nur kostenlos vertrieben werden, wenn sie unverändert oder geringfügig verändert sind und keine Produktinhalte begleiten, auf die die Bedingungen 2.3., 3.2. und 3.3. zutreffen.

2.3. Produktinhalte entsprechend der Bedingung 2.1. dürfen nur dann zu jedem Preis vertrieben werden, falls sie oder Teile davon bedeutend verändert worden sind.

No-mod Produktinhalte

3.1. Produktinhalte ohne den Bearbeitungsrecht, unabhängig vom Inhaltstyp mit Außnahme von Notekarten, dürfen nicht vertrieben werden wenn die Rechte des Nächstbesitzers *transfer und copy* sind. Diese Bedingung gilt für Inhalte, die separat vertrieben werden oder mit anderen Entwicklungen oder innerhalb diesen.

3.2. Wenn die Produktinhalte entsprechend der Bedingung 3.1. der Hauptgrund der Weitergabe ist, die Weitergabe darf nicht zu einem kleineren Preis stattfinden als der des adäquaten JFS Produkts.

3.3. Wenn die Produktinhalte entsprechend der Bedingung 3.1. nicht der Hauptgrund der Weitergabe ist, darf die Weitergabe zu jedem Preis stattfinden.

Mit anderen Worten...

Das TRP Paket enthällt Notekarten, Landmarken, Texturen und Objekte (mit weiteren Inhaltn), die fullperm sind. Diese und das fullperm Inhalt von Objekten müssen fullperm bleiben. Sie dürfen außerdem nur kostenlos weitergegeben werden, solange sie unverändert sind oder nur geringe Änderungen erfahren haben.

Geringe Änderungen sind etwa das Umbenennen von Inhalten, Korrektur von Schreibfehlern in den Skriptkommentaren oder Notekarten, ohne dass der Sinn des Textes geändert wird. Auch umbenennen von Variablen im Skirpt oder Vertauschen der Zeilen ohne dass dabei sie Semantik des Programs ändert. Solche Änderungen sind nicht wichtig genug um als *bedeutend* zu gelten.

Bedeutende Änderungen sind etwa graphische Aufbesserungen der Objekte, Erweiterung der Skriptfeatures oder Hinzufügen von eigenen Skripten und Dokumentationen. Nach solch einer Änderungen darf das geänderte Produkt zu jedem Preis vertrieben werden. Wurden außerdem nur Teile des Pakets geändert, gelten unveränderte Inhalte als begleitend für die Veränderten und können im gleichen Paket mitgegeben werden.

Was Sie neben den TRP Produktinhalten anbieten ist nicht Gegenstand dieser DLA, jedoch wenn Sie diese verkaufen, müssen Sie die in irgendeiner Form die Informatioonen über die Bezugsquelle von TRP mitgeben sowie über alle Veränderungen die Sie an TRP Inhalen vorgenommen haben, und Sie müssen eine unveränderte Kopie diese DLA mitgeben.

Es gibt nur eine Außnahme aus dieser Regel: Wenn Sie beispielweise ein unverpaktes Gerät wie ein Getrenkattachment an einem Automaten verkaufen, in diesem Fall könnte die Weitergabe der Bezugsinformation und der Lizensvereinbarung den Verkaufsprozess ruinieren. In diesem Fall darf das ausgelassen werden.

Die Lizensvereibarung versucht sicherzustellen, dass TRP (wie jedes andere Entwicklungsprodukt von JFS) kein Freeby wird und auf der anderen Seite Ihnen eine freie Entwicklung von Produkten erlaubt die mit TRP betrieben werden.

2.3 Einsatz von TRP

2.3.1 Auspacken

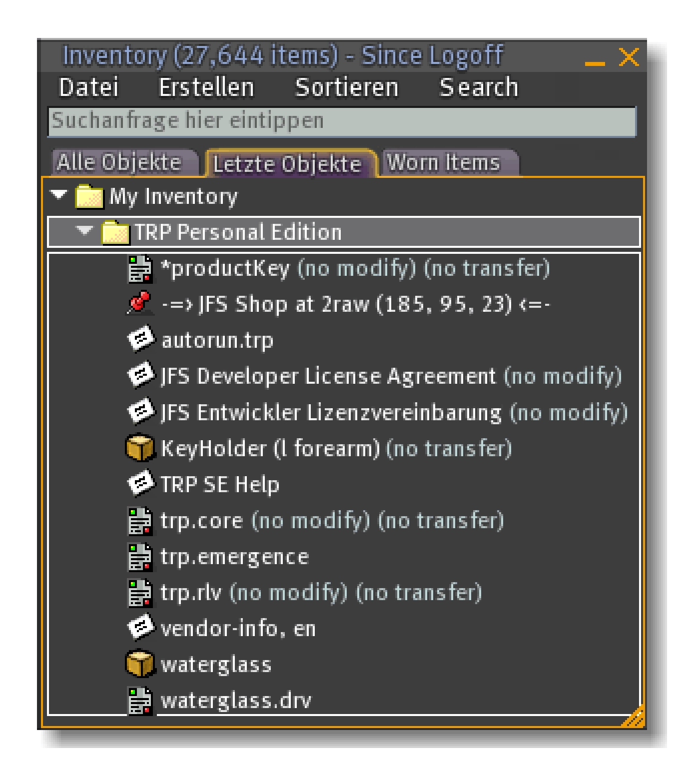


Figure 2.2: Inhalt des Produktpaket (ausgepakt)

Der einfachste Weg, ds Produktpaket zu enpacken ist über die Box selbst. Dann brauchen Sie sich um die vergessene Box nicht mehr zu kümmern. Schauen Sie sich nach einem Platz um, wo das Rezzen möglich ist und Skripte erlaubt sind, am besten Ihr Zuhause oder eine Sandbox.[5]

Nun, rezzen Sie die Produktbox. Ein Menü öffnet sich, bitte den button *Open* anklicken. Der Button *Update* ist zum Anfordern von Updates da, und der button *Remove* wird die Box aus der Welt entfernen, das ist an Dieser Stelle noch zu früh.

Nachdem ein Ordner übergeben wurde (bitte annehmen) erscheint ein weiteres Menü, dort bitte den Button *Remove* anklicken. Das entfernt die Box, da wir sie nicht mehr brauchen. Haben Sie die Menüs ignoriert, können Sie die Box nochmal anklicken, das erste Menü erscheint wo Sie die Box entfernen lassen können. Tun Sie das nicht, löscht sich die Box nach 30 Minuten wobei sie in der Mitte eine Erinnerungsnachricht an Sie sendet.

[5]Ob ein Platz Skripte erlaubt erfahren Sie im Landinfo.

Das war's, der Inhalt der Produktbox finden Sie als Ordner ih Ihrem Inventar, Abbildung 2.2. Bitte löschen Sie die originelle Produktbox nicht, es ist eine gute Sicherheitskopie und auch eine Möglichkeit, ein Update einzufordern.

2.3.2 Installation

Wir betrachten die Installation von TRP anhand des Beispielattachments das geliefert wurde, gemieint sind die Elemente '`waterglass`', '`autorun.trp`' und '`waterglass.drv`', die wir gerade ins Verzeichnis *examples* verschoben haben, Abschnitt **??**.

2.3.2.1 Installation

Die Einrichtung eines TRP-Systams sollte in dieser Reihenfolge stattfinden:

1. Das *Systemträger*. In unserem Fall ist das '`waterglass`' - Objekt. Bitte rezzen sie es auf dem Boden, bearbeiten sie es und wechseln Sie zum Tab *Inhalt*.

2. Alle *tragbare Elemente* für die Transformation ins Geräteinventar ablegen. Das wären Körperteile wie Shape, Skin, Augen, Kleidung, Attachments. In unserem Fall gibt es keine dieser Elemente.

3. Alle *Treiberskripte* ins Geräteinventar ablegen: Nur den Skript '`waterglass.drv`' im unseren Fall.

4. Alle *Pluginskripte* ins Geräteinventar ablegen. Das sind die Skripte '`trp.rlv`', '`trp.emergence`', sie werden aber in unserem Fall nicht benötigt.

5. Den *Interpreterskript* '`trp.core`' ins Geräteinventar ablegen. Wir müssen eine Version mit passenden Besitzerrechten auswählen, z.B. die aus dem Ordner *no transfer*.

6. Alle *Programme*, d.h. Notekarten mit dem Suffix '`.trp`'. In unserem Fall ist das nur die Notekarte '`autorun.trp`'.

Hinweis: In dieser Reihenfolge wurde der Interpreterskript vor der Notekarte '`autorun.trp`' installiert. Dadurch startet der Interpreter die Ausführung nicht sofort, sondern wenn das gerät aufgenopmmen und nue gerätzt oder aufgesetzt wird.

Ist es aber erwünscht dass der Interpreter die Note '`autorun.trp`' schon beim Installieren startet, muss die Note nach anderen Programnoten aber vor dem Interpreterskript installiert werden, die Schritte 5. und 6. sind zu vertauschen.

2.3.3 Update

Der JFS Updater benutzt ein spezielles Updatesystem. Früher war er in jeden JFS Vendor integriert, jetzt inform eines Updateorbs inworld getrennt installiert. Der Updateorb kommuniziert mit dem Skript '`*productKey`', der in jeder Produktbox vorliegt. Der Skript ist in der Lage, eine gültige Anfrage für das Produktupdate an den Orb zu senden. Er ist tatsächlich die einzige möglichkeit, ein Update zu erhalten: Nur wenn man ein Produkt erworben hat, besitzt man den Skript und nur wenn man den Skript besitzt, kann man auch das Update anzufordern.

Der '`*productKey`' Skript ist bereits in der Produktbox so eingerichtet, dass man damit die Updates abholen kann, einfach nur die Box in der Nähe des Updateorbs rezzen. Um es komfortabler zu machen und um das herumschleppen der Boxen zu vermeiden, steht ein sogenannter Keyholder zur Verfügung (ein Schlüsselhalter). Dieser kann bis zu 12 Schlüssel verwalten, d.h. damit kann man Updates für bis zu 12 Produkte anfordern.

In jedem Fall muss man in der Nähe des Updateorbs sein, das inworld instaliert ist. Zur Zeit wo dieses Dokument verfasst ist, ist der Updateorb im Mainstore installiert, [6]. Da diese Orte sich ändern können, wirde ein Blogbeiatrag verfasst, der die Shops auflistet und angibt wo der Updateorb zu finden ist, [4].

Hinweis: Das Update ist einfach eine Nachlieferung des aktuellen TRP Pakets. Nach auspacken werden Sie möglicherweise die Vorbereitungsschritte erneut durchführen müssen, Abschnitt **??**.

2.3.3.1 Update via Produktbox

Wenn Sie die Produktbox oder einen anderen Prim benutzen wo der '`*productKey`' Skript installiert ist, dann. . .

1. Kommen Sie in die Nähe des Updateorbs (unter 10m Distanz)
2. Die Produktbox bitte rezzen, ein Menü erscheint.
    a) Wenn nicht, bitte anklicken
    b) Das *Update* button bitte anklicken
    c) Der Updateorb wird das Update ausliefern

2.3.3.2 Update via Keyholder

Benutzen Sie den Keyholder, dann…

1. Kommen Sie in die Nähe des Updateorbs (unter 10m Distanz)
2. Tragen Sie den Keyholder und klicken Sie ihn bitte an, ein Menü erscheint
    a) Wenn der TRP Schlüssel nicht ausgewählt ist, bitte auswählen
    b) Nun bitte den *Update* Button anklicken
    c) Der Updateorb wird das Update ausliefern

Das war's auch schon :)

2.4 TRP Programbeispiel

```
1  # TRP-controlled device: Glass of Mineral Water
2  # Scenario: * intro
3  #           * drinking until empty
4  #           * interactive unwearing
5  #
6  # Version 1.0, $Revision: 1.3 $$Date: 2010/09/30 20:30:43 $
7
8  rename %fn%' glass
9
10 wait 3
11
12 whisper %an% takes a glass of cool drink
13
14 while not lm 15 this 0 isempty?
15     lm this 0 drink
16     wait 10+15
17 end
18
19 whisper //me is empty
20
21 if dialog 30 Detach, Drop, Place; What to do with empty glas?
22     lm this 0 detachme
23 else
24     whisper %fn% drops the glass on the floor
25     lm this 0 dropme
26 else
27     whisper %fn% puts the empty glass back on the table
28     lm this 0 placeme
29 default
30     lm this 0 detachme
31 end
32
33 clean script
```

Figure 2.3: TRP Program, beiliegende Beispielnotekarte

2.5 TRP Programmiersprache

TRP ist in erster Linie ein Sprachinterpreter. Dieser Abschnitt befasst sich seiner Programmiersprache[6]. Dafür werden wir kurz ein fertiges Program anschauen, die syntaxische Eigenschaften von TRP abreißen und schließlich jedes Befehl detailiert definieren.

2.5.1 Syntax

Wir untersuchen das Program das dem TRP Paket beiliegt, Abbildung 2.3.

2.5.1.1 Allgemeine Eigenschaften

- Zeichen '`#`' und der Text bis zum Zeilenende ist ein Kommentar
- Alle andere Wörter sind *Befehle* mit keinem, einem oder mehr *Parameter*
- Sekundäre Befehle eines größeren Konstrukts werden *Operatoren* bezeichnet
- Befehlsparameter werden zumeist durch Leerzeichen getrennt und nie eingeklammert
- Jedes Befehl wird in der eigenen Zeile notiert, etwa `else if` gibt es nicht
- *Zeileneinrückungen* sind irrelevant, dafür machen den Code lesbar
- Befehlesnamen unterscheiden Schreibweise nicht, '`else`' entspricht '`ELSE`'
- Es gibt keine Variablen oder benutzerdefinierte Funktionen

2.5.1.2 Erklärung des Code

Gehen wir nun den Code, Abbildung 2.3, Zeile für Zeile.

Der kurze Kommentarblock, Zeilen 1 bis 6, ist eine übliche Präambel, zum Einführen ins Code. Da ist nichts zu erklären. Jedoch eine Bemerkung: Leere Zeilen und reine Kommentarzilen verbrauchen Skriptzeit fürs Lesen, tun aber nichts. In realen Programmen sollten solche Zeilen beser vermieden werden: Kommentare sollten in den Codezeilen untergebracht werden und Codezeilen hintereinander stehen. Diese Praxis wird den Skriptlag verringern, aber natürlich auch die Lesbarkeit.

Die Zeile 6 sagt dem Interpreter, das Gerät unter Verwendung des Benutzer-Vornamen zu umbenennen. Nimmt etwa Jenna Felton das Getränk in die Hand, ist sein Name dann "*Jenna's glass*".

[6]Böse Zungen würden eher von einer Skriptsprache reden

Der `wait` Befehl in der Zeile 10 veranlasst eine Pause von genau 3 Sekunden (Zeit für Viewer anderer Benutzer das Getränk zu rezzen).

Der Befehl `whisper`, Zeile 12 sagt dem Interpreter den angegebenen Text auszuflüstern, wobei diesmal der volle Name des Benutzer verwendet ist, hält weiterhin Jenna Felton das Glass, ist der ausgeflüsterte Text dann "*Jenna Felton takes a glass of cool drink*".

Nun folgt die erste Interaktion, die `while` - Schleife an den Zeilen 14 bis 17. Der `while` Befehl nimmt die Bedingung "`lm 15 this 0 isempty?`" als Parameter und führt die Schleifenanweisungen (Zeilen 15 und 16) solange aus wie die Bedingung nicht erfüllt ist.

Die Bedingung bedeutet wiederum dies: Versende den Befehl "`isempty?`" via Linknachricht an den Gerätetreiber und erwarte seine Antwort. Was die Bedeutung dieser Nachricht ist, entscheidet der Treiber. In diesem Fall ist es eine Abfrage ob das Glass schon leer ist oder noch nicht. Solange die Antwort negativ ist, ist das Glass nicht leer und der Schleifencode wird ausgeführt.

Der Schleifencode sind die Zeilen 15 und 16. Die Zeile 15 veranlast den Interpreter, den Befehl "`drink`" als Linknachricht zu versenden (der Gerätetreiber wird dies als ein Befehl interpretieren, den trinkenden Code auszuführen) und die Zeile 16 ist ein Befehl, eine zufällige Zeit zwischen 10 und 25 Sekunden zu warten. Danach wird die Schleifenbedingung erneut geprüft.

Sobald das Glass leer wird, die Schleife ist verlassen und der Interpreter liest den `whisper` Befehl in der Zeile 19. Der ausgeflüsterte Text kündigt an dass das Glass leer ist. Der Doppelslash wird zu einem Einzelslash übersetzt, der zusammen mit dem Textanfang '`me`' eine Objektemote produziert.

Nun folgt der nächste Interaktive Teil, in diesem Fall mit dem Benutzer, und zwar die `if` - Abfrage auf den Zeilen 21 bis 31. Zunächst liest der Interpreter den Befehl `if` der sagt: Prüfe die Bedingung und entscheide die weitere Aktion. Die Bedingung ist aber der `dialog` - Operator. Dieser öffnet das Dialog mit drei Buttons '*Detach*', '*Drop*', '*Place*' und einer Frage. Die Zahl 30 bedeutet, dem Benutzer sind 30 Sekunden eingeräumt, einen der Buttons zu betätigen.

Nun, der Benutzer könnte den ersten, den zweiten, den dritten Button im Menü klicken, oder es ignorieren. Davon abhängig führt der Interpreter den Code vor dem ersten `else` Operator, vor dem zweiten `else` Operator, vor dem `default` Operator, oder danach. Der Code in diesen Zweigen veranlasst den Interpreter einen der Befehle "`detachme`", "`dropme`", oder "`placeme`" via Linknachricht zu versenden, so dass der Treiber diese Aktion ausführen kann, und in zwei Fällen die Aktion auch in chat kommentiert wird.

Unabhängig welcher Zweig zur Ausführung kam, der Interpreter setzt diese mit dem Operator `end` auf der Zeile 31 fort. Da bleibt nur ein Befehl auf der Zeile 33 übrig:

Der Befehl `clean` mit dem Parameter `script` stoppt die Ausführung und entfernt die TRP Skripte, macht dadurch das Getränk nur einmalig nutzbar.

2.5.1.3 Syntaxvorlagen

Um die Vorstellung einzelnen Befehle einfacher zu machen, wurde in diesem Dokument auf Syntaxvorlagen zurückgegriffen. Es sind insgesamt vier: Befehlsplatzhalter, optionale Elemente, Elementlisten und Aufzählungen.

2.5.1.4 Befehlsplatzhalter

```
1 think <text>
```

In spitzen Klammern werden Befehlsplatzhalter notiert. Sie werden in einem tatsächlichen Befehl durch einen anderen text ersetzt. Was genau den Platzhalter ersetzt, wird durch die Semantik des Befehls `think` fest gelegt. Zum Beispiel kann auf diese Definition auch dieser Befehl zutreffen:

```
1 think Hello World!
```

2.5.1.5 Optionale Elemente

```
1 whisper [/<channel>] <text>
```

Eckige Klammer kennzeichnen ein optionales Element. Im obigen Beispiel sind zwei Platzhalter in Verwendung, der optionale `channel` und der obligatorische `text`. Der Slash gehört zum Channelplatzhalter, muss also mit ausgelassen werden. Dieser Definition genügen diese beide Befehle:

```
1 whisper Hello World!
2 whisper /100 Hello World!
```

In der Zeile 1 ist der Channel ausgelassen, in der Zeile 2 jedoch notiert.

2.5.1.6 Elementlisten

```
1 (, <button> )+
```

Eine Liste von ännlichen Elemente ist definiert durch runde Klammer, Trennzeichen und Quantorzeichen. im obigen Beispiel ist das eine Liste von Platzhaltern, vermutlich Buttons, das Trennzeichen ist ein Komma und das Quantorzeichen ist der Plus. Dieser Quantor Bedeutet, die Liste muss aus mindestens einem Button bestehen. Dieser Definition entspricht zum Beispiel diese Liste "`Apfel, Birne, Pflaume`".

Das Trenzeichen darf nur zwischen den Listenelementen vorkommen. Das Pluszeichen es ist das einzige Quantorzeichen das in dieser Dokumentation verwendet wird. Gebräuchlich sind auch das Sternzeichen '`*`' (die Liste darf leer sein) und das Fragezeichen '`?`' (höchstens ein Button in der Liste), diese kann man durch eine optionale Liste oder optionales Element darstellen.

2.5.1.7 Aufzählungen

```
command (value1 | value2 | value3) option
```

Aufzählungen definieren eine Auswahl von einer vorgegebenen Liste. Diese wird in runden Klammern eingeschlossen, mit dem Vertikalstrich als Trennzeichen und ohne Quantorzeichen. Die Beispieldefinition wird etwa zu diesem Befehl aufgelöst: "`command value2 option`".

2.5.1.8 Leerzeichen

Normalerweise dürfen Leerzeichen um die Elemente auftreten, dann sind Leerzeichen auch im Definitionstext vorhanden, etwa so:

```
say [/<channel>] <text>
```

Dieser Definition entspricht der Befehl "`say /100 Welcome`" und auch "`say     /100    Welcome`". Wo Leerzeichen nicht erlaubt sind, ist die Definition ohne angegeben, etwa so:

```
@(,<commmand>[:<option>[=<parameter>]])+
```

Der String "`@cmd:opt=par,cmd2:opt2=par2`" ist hiermit erlaubt, der String "`@ cmd:opt=par, cmd2:opt2 = par2`" aber nicht.

2.5.2 Kernbefehle

Das sind Einzeiler die vom Kernskript ausgeführt werden.

2.5.2.1 Identitätsbefehl: RENAME

```
rename [<name>]
```

Der Befehl `rename` vergibt dem Gerät einen anderen Namen, der im Chat erscheint. Ist der Parameter nicht angegeben, wird der Originalname wiederhergestellt. Der ersetzende Text unterstützt auch Textplatzhalter, Tabelle 2.1. Beispielweise:

`%an%`	wird durch den vollen Benutzernamen ersetzt, z.B. *Jenna Felton*
`%fn%`	wird durch den Vornamen des Benutzers ersetzt, z.B. *Jenna*
`%ln%`	wird durch den Nachnamen des Benutzers ersetzt, z.B. *Felton*
`\n`	wird durch den Zeilenumbruch ersetzt (für Namen nicht empfolen)

Table 2.1: Text placeholders

```
rename %fn%'s glass
```

Gehört das Glass zu Jenna Felton, dann bekommt es den Chatnamen "*Jenna's glass*".

2.5.2.2 Chatbefehle: SHOUT, SAY, WHISPER, THINK, IM

```
shout [/<channel>] <text>
say [/<channel>] <text>
whisper [/<channel>] <text>
think <text>
im <text>
```

TRP unterstützt diese fünf Chatbefehle. Alle unterstützen Textplatzhalter, Tabelle 2.1. Die Befehle `shout`, `say` und `whisper` entsprechen den Ruf-, Sprech- und Flustermeldungen auf der Chatleiste des Viewers. Diese Meldungen kommen vom Objekt und sind dunkelgrün im Chatverlauf. Diese Befehle erlauben angabe des Chat Channel für die meldung, wobei auch der negative Channel möglich ist:

```
shout Hello there!!!
say /100 can you hear me?
whisper /-23 no, you can't :)
```

Sollte der Befehlstext mit einem Slash beginnen und es ist kein Channel angegeben, ein Doppelslash muss benutzt werden, da ein Einzelslash am Anfang den Channel vorgibt. Beispiel:

```
say //me is empty. %fn% feels much better now...
```

Der Befehl `think` produziert eine private Besitzermeldung. Sie ist gelb im Chatverlauf und erzeugt auch Chatpartikel. Der Befehl `im` versendet eine IM an den Besitzer. Diese Meldung ist hellgrün im Chatverlauf, der Befehl verzögert den Interpreter um 2 Sekunden, erzeugt keine Chatpartikel.

2.5.2.3 Dialognachricht: MESSAGE

```
1 message <text>
```

Der Befehl `message` öffnet für den Besitzer ein blaues Dialogfenster mit der angegebenen Nachricht und dem einzigen *Ok* Button. Der Befehl verzögert nicht, öffnet keine Listener (lagarm), und beachtet nicht ob der Button angeklickt wurde. Der Text unterstützt auch die Textplatzhalter, Tabelle 1.1. Obwohl dieser Befehl schnel ist und die Aufmerksamkeit wie nichts anderes, sollte man damit sparsam umgehen, da man bei vielen Dialogs den Überblick verliert. Beispiel:

```
1 message WARNING\nPlease read my messages carefully!
```

2.5.2.4 Linknachricht: LM

```
1 lm (<link> | root | this | all) <number> <text>
```

Der Befehl `lm` versendet eine Linknachricht die vom Gerätetreiber empfangen werden kann. Beispielweise im Bispiel 2.3 wurde so die Trinkaktion und Ablegeaktionen des Gerätetreiber ausgelöst.

Der erste Parameter gibt das Ziel der Nachricht an. Es kann entweder eine Zahl sein, dann ist das die Primnummer im Linkset[7] oder einer der drei Namen. Der Name '`root`' bedeutet das Ziel ist Rootprim, der Name '`this`' bedeutet das Ziel ist der Installationprim des Interpreter und der Name '`all`' bedeutet die Nachricht muss an den ganzen Linkset gehen.

Der Parameter `number` ist der nummerische Parameter der Linknachricht. Jede Nummer ist möglich außer `-2018160314`. Der Parameter `text` ist der Stringparrameter des versendeten Nachricht. Hier sind auch Textplatzhalter erlaubt, Tabelle 2.1. Der Keyparamter der Nachricht ist immer `NULL_KEY`. Beispiel:

```
1 lm this 0 take drink
```

Dieser Befehl führt zum Ausführen dieses LSL Befehls:

```
llMessageLinked(LINK_THIS, 0, "take drink", NULL_KEY);
```

[7] Das Rootprim hat die Nummer 1, andere Prims eine größere

Anmerkung: Bitte den Unterschied beachten: Der Befehl `im` versendet eine *IM* und der Befehl `lm` eine *Linknachricht*.

2.5.2.5 Pausebefehl: WAIT

```
1 wait [[[<day>:]<hour>:]<min>:]<sec>
```

Der Befehl `wait` hält die Ausführung für die angegebene Zeit an. Die Zeit ist in Tagen, Stunden, Minuten und Sekunden angegeben, wobei von Tagen zu Minuten ausgelassen werden kann. Jede Komponente kann entweder eine feste Zahl sein oder eine Zufallszahl mithilfe des Plusoperators:

```
1 wait 40              # Warte 40 Sek
2 wait 3:40            # 3 Min, 40 Sec
3 wait 12:3:40         # 12 Std, 3 Min, 40 Sec
4 wait 1:12:3:40       # 1 Tag, 12 Std, 3 Min, 40 Sec
5 wait 10+30           # Zwischen 10 und 40 Sec
6 wait 10+2:3:10+30    # Zwischen 10 Std, 3 Min, 10 Sec
7                      # und 12 Std, 3 Min, 40 Sec
```

2.5.2.6 Ausführungswechsel: START

```
1 start <name>
```

Der Befehl `start` wechselt zur Ausführung des angegebenen Programs. Es wird die Notekarte mit Namen ‘`<name>.trp`’ aufgesucht. Wenn gefunden und als *ausführbar* erkannt, wird zur Ausführung des angegebenen Program gewechselt. Z.B. wird so die Notekarte ‘`strawberry.trp`’ ausgeführt:

```
1 start strawberry
```

- Eine Note ist ausführbar, wenn sie einen Inhalt hat und als fulperm vorliegt.
- Nach Abschluss einer Ausführung wird nicht zum Program zurückgekehrt das den Kontextwechsel veranlasst hat.
- Das ausführbare Program ‘`autorun`’ ist automatisch gestartet:
  - nach Reset der TRP Skripte
  - wenn das Gerät den Besitzer wechselt
  - beim Rezzen wenn das program noch nicht ausgeführt wird

2.5.2.7 Abschlussbefehl: CLEAN

```
clean [script]
```

Der Befehl `clean` terminiert die Programausführung.

- Ist der Parameter nicht angegeben, stoppt die Programausführung einfach.
- Ist der Parameter `script` angegeben, stoppt die Ausführung und die TRP Skripte sind aus dem gerät entfernt.

Anwendung des Parameters macht das Gerät nicht-zurücksetzbar.

2.5.3 Flußkontrolle

Befehle dieser Gruppe führen Befehle bzw. Befehlsblöcke abhängig von definierten Voraussetzungen. Es gibt drei Befehle dieser Art: `if`, `while` und `for`.

2.5.3.1 IF - Konstrikt

```
if [not] <condition>
    # commands #
else
...
else
    # commands #
default
    # commands #
end
```

Der `if` - Konstrukt erlaubt es einen Anweisungsblock in Abhängigkeit von einer Bedingung auszuführen, wobei der Parameter die Art der Bedingung angibt.

2.5.3.2 Behandlung von Bedingungen in TRP

Übliche Programmiersprachen behandeln den `if` - Konstrukt als eine Auswahl eines Ausführungszweigs anhand einer Bedingung die entweder wahr oder falsch sein kann, die Bedingung $x < 4$ kann nur einer von zwei Wahrheitswerte annehmen. Viele Programmiersprachen besitzen auch eine weitere Auswahlmöglichkeit, wo der Wert einer Variable mit einer vorgegebenen Werteliste verglichen und je nach Ergebnis einer der mehreren Zweige ausgeführt wird.

Der Skriptspeicher in LSL ist nicht unbeschränkt, daher deckt der `if` - Konstrukt von TRP beide Operationen ab. Zu diesem Zweck werden die Bedingungen in TRP zu einer Nummer des Zweigs aufgelöst, der auszuführen ist. Diese Nummer gibt einfach an, wieviele `else` Operatoren zu überspringen sind bis der auszuführende Zweig gefunden ist.

Das bedeutet: Ist die Bedingung wahr, wird sie zu der Zweignummer 0 aufgelöst und der Interpreter führt die Befehle aus die direkt dem `if` Operator folgen bis der Operator `else` gefunden ist. Ist die Bedingung falsch, wird sie zur Nummer 1 aufgelöst und der Interpreter fängt die Ausführung nach dem ersten gefundenen `else` Operator. Wählt die Bedingung eine höhere Zweignummer aus, werden mehrere `else` Operatoren übersprungen.

Der `default` Operator leitet einen Anweisungszweig ein, der nur dann ausgeführt wird, wenn die Bedingung nicht zu einer Zweignummer aufgelöst werden konnte. Das kann zum Beispiel auftreten wenn zum Auflösen der Bedingung Benutzerantwort erwartet wird, die nicht rechtzeigig kam.

2.5.3.3 Negierte Bedingungen

Der `not` Operator negiert die Bedingung, d.h. *wahr* wird zu *falsch* und umgekehrt. Da die TRP Bedingungen zu einer Nummer aufgelöst werden, wandelt der Operator einfach die Nummer um, indem sie von 1 abgezogen wird:

- Die *wahre* Bedingung entspricht 0. Nach `not` wird sie zu 1, also *falsch*
- Die *falsche* Bedingung entspricht 1. Nach `not` wird sie zu 0, also *wahr*
- Eine Bedingung die zu einer Nummer größer 1 aufgelöst wird, wird durch den `not` Oprator negativ und der Interpreter findet keinen Zweig um die Bedingung zu behandeln
- Auf die Ausführbarkeit des Defaultzweigs hat der `not` Operator keinen Einfluss

2.5.3.4 Zufallszahlen: RND

```
1 (if | while) rnd <num>
```

Die `rnd` Bedingung nimmt eine Nummer zwischen 0 und 100 als Parameter, würfelt[8] und entscheidet sich für den Fall 0 (wahr) oder 1 (else). Die vorgegebene Zahl ist die Wahrscheilichkeit in Prozent für den wahr-Zweig. Diese Bedingung wird immer zu den Nummern 0 und 1 aufgelöst.

[8] Ja, TRP kann tatsächlich die Zufallszahlen

Um zum Beispiel das Wort ‘Apfel’ in 3 Fällen von 10 (30 Prozent) und das Wort ‘Birne’ in 7 Fällen von 10 (70 Prozent) auszuflüstern, schreiben wir diesen Code:

```
1  if rnd 30
2      whisper Apfel
3  else
4      whisper Birne
5  end
```

Ist eine Wahrscheinlichkeit kleiner 0 oder größer 100 angegeben, wird der Wert auf 0 bzw. 100 korrigiert. Bei 0 wird nie der erste Zweig ausgeführt, bei 100 nie der Zweite. Wird der `not` Operator verwendet, wird die Auswahl der Zweige invertiert.

Angenommen, wir wollen die Wörter ‘Apfel’, ‘Birne’, ‘Pflaume’, ‘Kirche’ mit 25% Wahrscheinlichkeit ausflüstern? Da wir immer nur zwischen zwei Fällen wählen können, verschachteln wir einfach die `if`s:

```
1  if rnd 50
2      if rnd 50
3          whisper Apfel
4      else
5          whisper Birne
6      end
7  else
8      if rnd 50
9          whisper Pflaume
10     else
11         whisper Kirche
12     end
13 end
```

2.5.3.5 Benutzerinteraktion: DIALOG

```
1  (if | while) dialog <time> (, <button>)+ ; <text>
```

Die Bedigung `if` ist für die Interaktion mit dem Benutzer gedacht. Die Bedingung nimmt als Parameter eine Zahl zwischen 15 und 60, eine kommaseparierte Liste von Buttonnamen, die durch den Semikolon von einem Text getrennt ist. Der Text wiederum erlaubt die Verwendung von Textplatzhalter, Tabelle 2.1. Ist die Nummer kleiner 15 oder größer 60 wird sie auf 15 bzw. 60 korrigiert.

Zum Auflösen der Bedingung wird dem Benutzer ein blaues Menü geöffnet mit den angegebenen Buttons und der Textnachricht und auf seine antwort gewartet. Die Bedingung wird genau auf die Nummer des angeklikten Buttons aufgelöst.[9] Die Zahl

[9] Der erste Button in der Liste hat die Nummer 0

gibt die Wartezeit in Sekunden an, ist sie abgelaufen, gilt die Bedingung als nicht aufgelöst. Beispielweise so könnten wir den Benutzer bitten, die Geschmaksrichtung seines Getränk zu wählen:

```
1  if dialog 45 Apfel, Birne, Pflaume, Kirche; Was mögen Sie denn?
2      whisper Oh, Sie mögen Apfel
3  else
4      whisper Birne ist der beste Geschmack
5  else
6      whisper Pflaume schmeck auch gut
7  else
8      whisper mmmh, Kirrrsche...
9  default
10     whisper Sie können sich wirklich nicht entscheiden?
11 end
```

Dieser Konstrukt wird ein Dialog öffnen und dem Benutzer 45 Sekunden fürs Antworten einräumen. Merken Sie die Verwendung des `default` Operators der das Timeout abfängt?

2.5.3.6 Geräteinteraktion: LM

```
1  (if | while) lm <time> <link> <number> <text>
```

Die `lm` - Bedingung erlaubt eine Interaktion mit dem Gerätetreiber. Die Syntax erweitert die des `lm` Befehls um eine Zeitangabe. Es wird eine Linknachricht versendet anhand der restlichen Parametern, und eine Antwort des Gerätetreibers erwartet. Die Antwort muss eine Linknachricht sein, mit denselben nummerischen Parmameter und jedem String außer `NULL_KEY` als Schlüsselparameter. Die Bedingung wird auf die Zahl aufgelöst, die im Stringparameter der Antwort angegeben ist. Ist der Stringparameter leer, gilt die Bedingung als nicht aufgelost.

So lassen wir den Gerätetreiber selbst die Geschmaksrichtung wählen:

```
1  if lm 15 this 10 select:Apfel,Birne,Pflaume,Kirsche
2      whisper meine Geschmaksrichtung ist: Apfel
3  else
4      whisper meine Geschmaksrichtung ist: Birne
5  else
6      whisper meine Geschmaksrichtung ist: Pflaume
7  else
8      whisper meine Geschmaksrichtung ist: Kirsche
9  default
10     whisper Wie langweilig, keine Geschmaksrictung
11 end
```

Was Passiert hier. Um die Bedingung aufzulösen versendet der Interpreter eine Linknachricht mittels Ausführen dieses LSL Befehls

```
llMessageLinked(LINK_THIS, 10, "select:Apfel,Birne,Pflaume,Kirsche", NULL_KEY);
```

d.h. es wurde der TRP Befehl "`lm this 10 select:Apfel,Birne,Pflaume,Kirsche`" ausgeführt. Nun startet der Interpreter einen Timer für 15 Sekunden und wartet darauf dass eine Linknachricht mit 10 als nummerischer Parameter und nicht `NULL_KEY` als Schlüssel ankommt, das wird die Antwort sein. Angenommen wählt der Gerätetreiber 'Pflaume', dann muss er die Zahl 2 zurückgeben (die Antwort startet mit 0), also führt er diesen LSL Befehl aus:

```
llMessageLinked(LINK_THIS, 10, "2", "");
```

Diese Antwort muss innerhalb der angegebenen Zeit (15 Sekunden) ankommen, sonst gilt die Bedingung als nicht aufgelöst. Die Zeitangabe kann 0 oder negativ sein, dann hält der Interpreter solange bis die Antwort ankommt. Das eröffnet interessante Einsatzmöglichkeiten, jedoch nicht vergessen eine Antwortnachricht zu senden, sonst hängt der Interpreter.

2.5.3.7 Bedingte Schleife: WHILE

```
1 while [not] <condition>
2     # commands #
3 end
```

Die `while` - Schleife funktioniert wie in anderen Programmierschprachen. Sie führt einen Anweisungsblock solange aus wie die Bedingung wahr ist. Wird der `not` Operator benutzt, so wird der Anweisungsblock solange ausgeführt bis die Bedingung wahr wird. Eine TRP Bedingung ist wahr wenn sie zu 0 aufgelöst wird. Beispiel:

```
1 while rnd 75
2     whisper Hello there!
3     wait 2
4 end
```

Dieser Code wird den Gruß mit Wahrscheinlichkeit von 75% ausflüstern, und falls ausgeflüstert die Bedingung erneut prüfen, also es kann dann mit der Wahrscheinlichkeit von 75% zum erneuten Gruss kommen.

2.5.3.8 Zählerschleife: FOR

```
1 for <times>
2     # commands #
3 end
```

Die Zählerschleife führt den Anweisungsblock eine bestimmte Anzahl aus. Diese Anzahl kann entweder eine genaue Zahl sein oder via Plusoperators eine Zufallszahl im bestimmten Bereich nagegeben werden. Einfaches Beispiel:

```
for 5
    whisper Sie hören mich genau 5 mal
end

for 5+3
    whisper Sie hören mich zwischen 5 und 8 mal
end
```

2.5.4 Einführung in RLV API (RLVa)

RLV ist der Name eines speziellen SL Viewers und steht nun für die Familie von Viewern die eine besondere API unterstützen, die wiederum unter [9] spezifiziert ist. Schauen Sie bitte dort wenn Sie eine aktuelle und vollständige Beschreibung suchen. Diese hier zu wiederholen würde die Zitat nicht aktuell machen, und die Größe dieses Dokuments verdoppeln, daher wird die API nur kurz vorgestellt.

Über RLV selbst bitte im Entwicklerblog lesen, [5], an dieser Stelle nur soweit: Dieser Viewer wurde entwickelt um die immersive Erfahrung in SL zu erhöhen. Um das zu erreichen müssen Skripte eine Kontrolle über bestimmte Funktionen des Viewrs erlangen, was mit dem Originalviewer nicht möglich ist.

Nun, Skripte wrden auf der Sim ausgeführt und der Viewer ist ein local laufendes Program. Beide Programme müssen kommunizieren können. Die Eleganz des RLV Protokols liegt im Benutzen eines Kommunikationswegs den LL bereits implementiert hat: Besitzernachrichten richtung Viewer und Chatnachrichten richtung Skripte.

Besitzernachrichten[10] sind gelb im Chatverlauf, der Viewer empfängt sie nur wenn sie vom eigenen Objekt kommen. Der regulre Viewer würde sie alle anzeigen. Der RLV führt die Befehle aus und zeigt alle anderen normal an. Eine Besitzernachricht wird als Befehl angenommen wenn sie mit dem Zeichen '`@`' anfängt. Die Syntax der RLV Befehle ist einfach:

```
@(,<command>[[:<option>]=<param>])+
```

Das Zeichen '`@`' beginnt den Befehlsblock, der mit der NAchricht endet. Jeder Befehl beginnt mit dem Namen, gefolgt von optionalen Option und Parameter. Es gibt nur ein Befehl der diese auslässt.

[10]Versandt via `llOwnerSay`

2.5.4.1 Typen der RLV Befehle

Es gibt vier davon:

- Informationsabfrage: “`@<command>[:<option>]=<channel>`”
- Aktionen: “`@<name>[:<option>]=force`”
- Restrictions: “`@<name>[:<option>]=add`” oder “`@<name>[:<option>]=rem`”
- Clearbefehl: “`@clear`”

2.5.4.2 Informationsabfrage

Mit diesen Befehlen fragen die Skripte bestimte Informationen ab. Es gibt eine Reihe solcher Informationsabfragen, TRP bietet jedoch keine Befehle um sie direkt auszuführen sondern führt die Abfragen selbständig im Hintergrund, falls erforderlich.

Volständigkeitshalber werden wir hier nur eine Versionsabfrage betrachten, alle andere Abfragen funktioniren analog. Der Befehl ist ‘`version`’. Um es auszuführen verfährt der Skript wie fogt: Erst wird eine Channelnummer ausgewählt, sagen wir es ist 1234, und ein Listener gestartet um die Rückmeldung zu empfangen.

Nun versendet der Skript die Besitzernachricht ”`@version=1234`”. Der Viewer empfängt sie, ermittelt, dass es um eine Informationsabfrage handelt und versendet diese auf dem Channel 1234.

Der Skript empfängt und verarbeitet die Antwort und schließt den Listener. Implementiert der viewer die RLVa nicht, zeigt er die gelbe Nachricht an, ohne die Info zu senden. Der Skript empfängt keine Antwort in der festgelegten Zeit und schließt den Listener mit Erkenntnis, RLVa wird nicht unterstützt.

2.5.4.3 Clearbefehl

Es ist der einzige Befehl der weder Option noch Parameter besitzt. Der Effekt davon ist, alle RLV Restrickions sind entfernt die durch Skripte im selben Objekt gesetzt wurden. Der TRP Interpreter versendet diese Nachricht zweimal: Wenn das Program gestartet ist und wenn es terminiert ist wia des `clean` Befehls, Abschnitt 2.5.2.7.

Es gibt kein Befehl in TRP um diese Nachricht direkt zu versenden, allerdings versendet der `think` Befehl Besitzernachrichten, also geht diese Kommandozeile (nicht empfehlensert, da der RLV Plugin nicht involviert ist):

```
1 think @clear
```

2.5.4.4 Aktionen

```
@<name>[:<option>]=force|
```

Aktionen emulieren das Anklicken von Buttons auf der Vieweroberläche oder in Dialogfenstern. Sie ändern bestimmte Avatareigenschaften, produzieren aber keine permanente Änderungen an Viewerfunktionen.

Aktionen sind RLV Befehle mit dem Parameter `force`. Wichtigste davon sind in der Tabelle 2.2 angegeben. Einige erlaubt Angabe von Optionen, welchen Wert sie aufnehmen, gibt die Tabelle 2.5 an.

Aktion	Effekt
`sit:<UUID>`	Sitzen an einem Objekt
`unsit`	Erzwingt das Aufstehen
`tpto:<X>/<Y>/<Z>`	Teleport des Benutzers
`remoutfit[:<part>]`	Erzwingt das Ausziehen von Kleidung
`detach[:name]`	Erzwungenes Abldgen eines Attachments
`attach:<folder1/.../folderN>`	Anlegen von Objekte im verteilten Ordner
`attachall:<folder1/.../folderN>`	Dasselbe, rekursiv
`detach:<folder>`	Ablegen von Objekte im verteilten Ordner
`detachall:<folder1/.../folderN>`	Dasselbe, rekursiv
`detachme`	Ablegen des Attachments das befielt
`setrot:<angle>`	Avatar drehen
`setdebug_<setting>:<value>`	Debug Einstellungen ändern
`setenv_<setting>:<value>`	Umgebungseinstellungen ändern

Table 2.2: RLV Aktionen und deren Effekt

Nun, wie wird eine Aktion ausgelöst. Angenommen wir wollen dass der Avatar aufsteht. Die Aktion dazu ist '`unsit`' ohne Optionen. Der Skript wird daher einfach die Besitzernachricht "`@unsit=force`" absenden, der RLV versteht: Der Skript klickt den Aufstehen-Button an.

Ein anderes, komplesteres Beispiel ist die Aktion '`detach[:name]`'. Ein Blick in die API erklärt: Ist die Option nicht verwendet, diese Aktion legt alle Attachments ab, ist ein Attachmentpoint angegeben, wird nur Attachment abgelegt das dort angehängt ist. Mittels der Aktion der Skript klickt das Attachment an und dann den 'Abnehmen' Button.

2.5.4.5 Restrictions

```
@<name>[:<option>]=add
@<name>[:<option>]=rem
```

Eine Restriction ist eine permanennte Deaktivierung einer bestimmten Fähigkeit des Viewers[11]. Zum beseren Verständniss kann man die Restriktions als eine Seite unter Einstellungen vorstellen mit Checkboxen: Fliegen aus, Inventar verbergen, Minimap verbergen, und so weiter. Diese Seite ist aber nicht für den Benutzer sichtbar sondern für Skripte.

Da die restriktions permanennt sind, gibt es zwei Parameter, der Parameter `add` setzt die Restriktion, der Parameter `rem` entfernt sie. Einige wichtige Restricktions sind in den Tabellen 2.3, 2.4 aufgelistet und mögliche Optionenwerte gibt die Tabelle 2.5 an.

Angenommen der Skript möchte das Fliegen verhindern. Die Restriction dazu ist '`fly`', zum setzen wird der Parameter '`add`' verwendet. Also versendet der Skrit die Besitzernachricht "`@fly=add`". Diese Nachricht sagt dem Viewer: Öffne diese spezielle Seite und setze einen Hächken beim 'Fliegen aus'. Ab jetzt ist der Fliegen Button inaktiv, der Avatar kann nicht einmal im Gottmodus fliegen. Bis er ausloggt oder der Skript die Besitzernachricht "`@fly=rem`" versendet hat.

Neben Restrictions gibt es auch Ausnahmen, sie werden auf dieselbe Weise gesetzt oder entfernt wie Restrictions. Wenn gesetzt, sie beschränken die Wirkung von Restrictions auf bestimmte Weise. Zum Beispiel führt die Restriction '`tplure`' dazu, dass jede TP-Abfrage vom Viewer automatisch abgewiesen wird. Nun möchte man aber, die TP-Anfrage von bestimmten Personen erlauben. Die zugehörige Ausnahme lautet '`tplure:<UUID>`', also werden nach der Besitzernachricht "`@tplure:eb66b7b7-7ddb-4c5a-95ad-bfeb9837ae29=add`" die TP-Anfragen nicht abgewiesen, falls sie von Jenna Felton kommen[12].

2.5.5 RLV Befehle

RLV Befehle sind Einzeiler, die via RLV Plugins behandelt werden.

2.5.5.1 Restriction management: ADD, REM

```
1 add (, <restr>)+
2 rem (, <restr>)+
```

Diese beide Befehle setzen und enfernen die RLV Restrictions und Ausnahmen, Tabellen 2.3, 2.4 und 2.5. Um, beispielweise die Restriction `unsit` zu setzen (Avatar kann nicht mehr aufstehen), aber die Restrictions `showinv`, `edit`, `rez` zu entfernen (Inventory, Editieren und Rezzen wieder aktiv) breuchen wir diese zwei Befehlszeilen:

[11] Der Zustand verbleibt bis das Objekt abgelegt ist oder der Viewer ausgeloggt wird
[12] ihre UUID wurde hier nähmlich verwendet

Restriction	Effekt
sendchat	Deaktiviert das Versenden von Chatnachrichten
chatshout	Verhindet das Rufen im Chat
chatnormal	Verhindet das Sagen im Chat
chatwhisper	Verhindet das Flüstern im Chat
emote	Ausnahme: Emotes nicht abschneiden
redirchat:<chan>	Leitet Chat auf private Channel
rediremote:<chan>	Dasgleiche mit Emotes
sendchannel[:<chan>]	Verhindert senden auf Privatchannel außer ...
recvchat	Verhindert das Hören im Chat
recvchat:<UUID>	Ausnahme: Avatar ... wird gehört
recvemote	Verhindert das Sehen von Emotes
recvemote:<UUID>	Ausnahme: Emotes von ... werden gehört
sendim	Verhindert das Versenden von IM's
sendim:<UUID>	Ausnahme: IM's senden an ... erlauben
recvim	Verhindert das Empfangen von IM's
recvim:<UUID>	Ausnahme: IM's empfangen von ... erlauben
showinv	Deaktiviert das Inventar
viewnote	Verhindert das Lesen von Notekarten
viewscript	Verhindert das Öffnen von Skripte
viewtexture	Verhindert das sehen von Texturen
edit	Verhindert das Bearbeiten von Objekte
rez	Verhindert das Rezzen von Objekte
tplm	Verhindert das Teleportieren per Landmarke
tploc	Verhindert das Teleportieren per Weltkarte
tplure	Weist TP-Anfragen ab
tplure:<UUID>	Ausnahme: TP-Anfragen von ... erlauben
accepttp[:<UUID>]	TP-Anfragen von ... automatisch annehmen
sittp	TP durch Sitzen auf 1.5m max beschrenken
sit	Verhindert das setzen auf Gegenständen
unsit	Verhindert das Aufstehen
fly	Deaktiviert das Fliegen
fartouch	Berühren auf 1.5m max beschrenken

Table 2.3: RLV Restrictions und deren Effekt, Teil 1

Restriction	Effekt
showworldmap	Deaktiviert die Weltkarte
showminimap	Deaktiviert die Minikarte
showloc	Verbirgt die Avatarposition
shownames	Deaktiviert das Sehen von Namen
showhovertextall	Verhindert das Sehen von Hovertexten
showhovertext:<UUID>	Dasselbe für bestimmte Objekte
showhovertexthud	Dasselbe für HUD-Objekte
showhovertextworld	Dasselbe für Objekte inworld
detach[:<name>]	Sperrt den Attachmentpoint
addattach[:<name>]	Sperrt den leeren Attachmentpoint
remattach[:<name>]	Sperrt den belegten Attachmentpoint
addoutfit[:<part>]	Verhindert das Tragen von Kleidung
remoutfit[:<part>]	Verhindert das Entfernen von Kleidung
setdebug	Verhindert das Ändern von Debugsettings
setenv	Verhindert das Ändern von Umgebungseinstellungen

Table 2.4: RLV Restrictions und deren Effekt, Teil 2

Action or Restriction	Option values
detach, addattach, remattach	chest, skull, left shoulder, right shoulder, left hand, right hand, left foot, right foot, spine, pelvis, mouth, chin, left ear, right ear, left eyeball, right eyeball, nose, r upper arm, r forearm, l upper arm, l forearm, right hip, r upper leg, r lower leg, left hip, l upper leg, l lower leg, stomach, left pec, right pec, center 2, top right, top,top left, center, bottom left, bottom, bottom right
addoutfit, remoutfit	gloves, jacket, pants, shirt, shoes, skirt, socks, underpants, undershirt, skin, eyes, hair, shape, alpha, tattoo
setdebug*, setenv*	Bitte in der RLVa schauen, [9]

Table 2.5: Optionswerte für Aktionen und Restrictions

```
1 add unsit
2 rem showinv,edit,rez
```

Der TRP Interpreter prüft gar nicht ob angegebene Restrictions ausführbar sind, sondern leitet sie an den Viewer weiter. Dieser zeigt dann die Fehlermeldung wenn der Befehl nicht ausführbar ist. Auf diese Weise der Interpreter ist bereit für Befehle die in Zukunft dem RLVa hinzugefügt werden.

2.5.5.2 Auslösung von Aktionen: FORCE

```
1 force (, <actn>)+
```

Dieser Befehl löst RLV Aktionen aus. Die Anwendung ist dem Setzen von Restrictions ännlich: Der Befehl nimmt eine Liste von Aktionen zum Auslösen. Um den Avatar zum Aufstehen zu zwingen, benötigt es die Aktion `unsit`. Diese Befehlszeile löst diese Aktion aus:

```
1 force unsit
```

Auch hier, es wird nicht geprüft ob die Aktion ausführbar ist sondern zum Viewer weitergeleitet. Damit ist der Interpreter bereit für Aktionen die in Zukunft dem API zugefügt werden.

2.5.5.3 Kleidnugsanagement: WEAR, UNWEAR

```
1 wear (, <name>)+
2 unwear (, <name>)+
```

Diese beide Befehle arbeiten mit Elementen im Objektinventar. Das erlaubt es, den Avataroutfit wesentlich einfacher zu managen. Grundsätzlich wird beim Lesen des `wear` Befehls das angegebene Element an den Avatar ganz normal übergeben und der Viewer angefordert, die übergebnene Elemente via RLV Aktionen anzuziehen. Deteils sind aber komplizierter.

Zum Einen, hat RLVa keine Möglichkeit, Objekte einzeln aufzusetzen, sondern arbeitet nur mit Ordner. Deshalb müssen die Elemente in Ordner übergeben werden. Übergabe einer gr0ßen Zahl von Ordnern ist belästigend, also werden alle `wear` Befehle gespeichert bis der letzte Befehl des Bloks gelesen ist, erst dann werden die Elemente in einem einzelnen *Container*ordner übergeben, und sein Inhalt aufgesetzt, sobald er im Avatarinventar gefunden ist.

Der `wear` Befehl blokiert: Der Container wird alle 15 Sekunden übergeben bis der Avatar diesen tatsächlich animmt und das Anziehen möglich wird. Erst dann wird die Ausführung vortgesetzt.

Der `unwear` Befehl funktioniert anders, es blokiert nichts, nichts wird gespeichert, die Elemente wie sie im Program stehen direkt abgenommen via RLV Aktionen.

Beide Befehle arbeiten mit jedem Typ tragbarer Sachen, man kann auch Korperteile wie Shape, Skin, Kleidung und Attachments in einem Befehl kombinieren. Da Körperteile nicht abgelegt werden können, werden sie als Parameter des `unwear` ignoriert.

Machen wir ein Beispiel. Angenommen ein Gerät soll den Avatar in eine Marmorstatue verwandeln. Dazu muss er ein Skin, Augen, Haare tragen, einen Krug und einen Sockel. Diese Gegenstände müssen im Objektinventar vorliegen, aufsetzen wird mit diesen Befehlszeilen vollzogen:

```
1 wear White Marble Skin, White Marble Eyes, White Marble Hair
2 wear White Marble Jug (l forearm)
3 wear White Marble Base Low (left foot)
```

Wird nach der Zeile 3 ein anderes Befehl als `wear` gefunden, wird dieser Befehlsblock ausgeführt, also Elemente in einem Container übergeben und sobald der Container im Inventarordner gefunden ist, eine RLV Aktion zum Aufsetzen lassen.

Wollen wir die Items ablegen, so reichen diese Befehlszeilen:

```
1 unwear White Marble Jug (l forearm)
2 unwear White Marble Base Low (left foot)
```

Sobald der Interpreter die Befehle liest, findet er wo die Elemente getragen sind und setzt entsprechende Aktionen ab.

Anforderungen

Die Befehle `wear` und `unwear` setzen folgende Anforderungen an Elemente im Inventar:

1. Jeder angegebene Element muss im Objektinventar sein und der Besitzer muss den *copy* Recht darauf haben. SL weigert sich no-copy Objekte in Ordner zu geben.
2. Bei Attachments, der Elementname muss den Namen des Attachmentpoints in runden Klammern angebben, etwa "Skirt (Pelvis)". Sonst kann der Interpreter den Attachmentpoint zum Ablegen nicht ermitteln.
3. Bei der Kleidung, der Elementname muss den Namen des Kleidungslayer angeben, ebenfalls in Klammern, etwa "Latex strings (underpants)". Sonst kann der Interpreter den Kleidungslayer zum Ausziehen nicht ermitteln.

Namen der Attachmentpoints und Kleidungslayer stehen in der Tabelle 2.6.

2.5.5.4 Kleidungsabnahme: STRIP

```
1 strip (, <part>)+
```

Dieser Befehl erlaubt es, Kleidung, Attachments und deren Grupen auf gleicher weise abzulegen, was die Aktionen vereinfacht. Der Befehl erwartet Namen der einzelnen Kleidungslayer, Attachment points oder Gruppen. Besipielweise kann man mit der folgenden Kommandozeile alle Objekte entfernen lassen die am HUD angehängt sind:

```
1 strip huds
```

Liest sie der Interpreter, produziert dieser eine lange Reihe von Aktionen für jeden Attachmentpoint. Die Tabelle 2.6 zeigt Namen der eizelnen Kleidungslayer und Attachmentpoints so wie die Grupennamen.

Gruppe	Kleidungslayer oder Att. Punkt
clothes	gloves, jacket, pants, shirt, shoes, skirt, socks, underpants, undershirt, alpha, tattoo
huds	center 2, top right, top, top left, center, bottom left, bottom, bottom right
head	skull, mouth, chin, left ear, right ear, left eyeball, right eyeball, nose
body	chest, left shoulder, right shoulder, left hand, r upper arm, r forearm, l upper arm, l forearm, right hip, r upper leg, r lower leg, left hip, l upper leg, l lower leg, stomach, left pec, right pec, right hand, left foot, right foot, spine, pelvis
objects	huds, head, body D.h. jedes individulle Element dieser Gruppen

Table 2.6: Gruppennamen und eizelne Elemente in der Gruppen

2.5.5.5 Teleportbefehl: TP2LM

```
1 tp2lm <name>
```

Dieser Befehl übernimmt den Namen einer Landmarke im Geräteinventar und erzwingt einen Teleport zur Zielposition. Wenn diese Stelle durch einen Landungspunkt gesichert ist, wird ein Sprung unternommen der den Avatar sehr nah an die Zielposition der Landmarke bringt. Beispiel:

```
1 tp2lm Dreamland of Protection
```

In diesem Fall ist der Landmarkname "*Dreamland of Protection*". Der Befehl ist blokierend: Fehlt die angegebene Landmakre, wird kein TP unternommen, sonst wird die Programausführung nur fortesetzt, wenn der Avatar an der Landmarkposition ankommt oder so nahe wie möglich gebracht wurde.

Anmerkung zu Teleports

Teleports in SL sind zeitweise abenteuerlich, bitte verwenden Sie diesen Befehl weise: Manchmal ist die Zielsim zu langsam beim Annehmen der Teleportanfrage, man kerkt das wenn das manuelle TP mit einer Fehlermeldung abgebrochen wird. Allerdings kann der TRP diese Meldung nicht empfangen und wird solange den Versuch unternehmen bis der Avatar teleportiert wurde. Das kann unter Umständen auch im Ausloggen es avatars enden. Lindens arbeiten zwar daran, aber wenn die Grid oder Zielsim überlastet sind, kann dieser Fehler trotzdem auftreten.

Anmerkung zum Warping

Erfolgte der Teleport, ist der Vorgang nicht immer abgeschlossen: Einige Plätze benutzen Landepunkte die den Avatar abfangen. Das mag für Malls oder Clubs von Vorteil sein aber nicht für RPs. Deshalb versucht der Interpreter den Avatar auch zur Zielposition zu warpen. Warping bedeutet: Der Avatar wird mit einer Geschwindigkeit zur Zielposition gepuscht, die hoch genug ist um feste Mauer zu durchbrechen[13].

Das Warping wird in einem der drei Fällen abgebrochen:

1. Der Warpingweg führt durch eine Mauer oder anderen Objekt das zu dick ist um durchbrochen zu werden. Der Interpräter gibt auf und lässt den Avatar vor der Mauer
2. Die Sim hat Schaden aktiv oder der Warpingweg führt durch eine Parzelle mit aktivierten Schaden. Kollision mit festen Gegenständen im Flug kann zum 'Töten' des Avatars führen, so dass er nachhause teleportiert wird. Um das zu vermeiden wird das Warpen abgebrochen und die RP setzt sich am Landepunkt fort oder schadenaktive Parzelle in die der Avatar eingeflogen hat.
3. Der Avatar hat fliegen aktiviert. Fliegende Avatare können nicht gepuscht werden, so dass der Avatar am Landepunkt gelassen wird

[13]Es wurde schon mal eine Geschwindigkeit von 150 m/s gemessen

Während das Vermeiden beider ersten Gründe nur durchs Planen der RP möglich ist, so dass der Warpweg keine feste Gegenstände oder Schadensaktive Parzellen kreuzt, kann man das Fliegen des teleportierenden Avatars vermeiden, indem man vor dem TP das Fliegen verbietet und nach dem TP wieder erlaubt. Das Beispielprogram wäre das:

```
add fly
tp2lm Dreamland of Protection
rem fly
```

2.5.6 RLV - Bedingungen

Ist der RLV Plugin installiert, werden auch zwei Bedingungsoperatoren weiteren unterstützt.

2.5.6.1 Prüfung auf: RLV

```
(if | while) [not] rlv
<constraint>
```

Die Bedingung `rlv` erwartet keine Parameter und wird zu 0 aufgelöst (wahr), wenn der Benutzer RLV-kompatiblen Viewer verwendet und zu 1 (falsch) sonst. Beispiel:

```
if rlv
    wear Marble-skin, Marble-eyes, marble-hair
    ...
else
    message kein RLV gefunden, Transformation geht nicht!
default
    message keine Antwort vom RLV Plugin
    think Möglicher Skriptcrash. Beende RLV Unterstützung
end
```

Prüfen Sie bitte immer auf vorhandensein von RLV, wenn Sie wenigstens die blokierende `wear` und `tp2lm` einsetzen wollen, um das Aufhängen des Programs zu verhindern.

2.5.6.2 Kleidungscheck: WORN

```
(if | while) [not] worn <part>
<constraint>
```

Die Bedingung `worn` nimmt den Namen eines Kleidungslayer, Attachment points oder Gruppe und wird zu 0 aufgelöst (wahr) wenn diese benutzt werden oder zu 1 (falsch) wenn sie frei sind. Der Parameter `part` ist zimlich derselbe wie der Parameter des `strip` Befehls, Tabelle 2.6.

Wie funktioniert die Bedingung:

- Ist der Parameter `part` ein Name eines bestimmten Kleidungslayer oder Attachmentpoints, dann wird die Bedingung zu 0 aufgelöst (wahr), wenn dieser belegt ist und zu 1 (falsch), wenn die angegebene Stelle frei ist.
- Ist der Parameter dagegen name einer Gruppe, dann wird die Bedingung zu 0 aufgelöst, wenn irgendeiner Element dieser Gruppe belegt ist und zu 1, wenn alle Elemente der Gruppe unbelegt sind.

Beispiel:

```
1 if worn clothes
2     whisper Schön, Sie tragen etwas. Nix zu tun
3 else
4     whisper Hey, Sie sind ja NAKT! Das ändern wir mal...
5     wear Jeans (pants), Shirt (shirt), Sneakers (shoes)
6 default
7     message Keine Ahnung ob Sie was an haben, hoffentlich doch
8 end
```

2.6 Script API

Dieser Abschnitt ist für Entwickler gedacht, die wissen wie man Linknachrichten in einem Skript vrwendet. Jeder Skript, der mit dem TRP Interpreter auf diese Weise interagiert wird als ein (Geräte)treiber betrachtet, ob er irgendwelche Funktionen des Geäts oder nicht. Es gibt zwei Kategorien von Nachrichten die der Interpreter unterstützt: *Kontrol*- und *interaktive* Nachrichten.

2.6.1 Kontrolnachrichten

Kontrolnachrichten sind durch Treiber oder den Interpreter initiiert und werden mittels Linknachrichten versendet mit diesen Parameter:

- Der Interpreter benutzt `LINK_THIS` als Linkziel.
- Der Nummerparameter ist -2018160314 (`TRPCN` in Zahlen).

2.6.1.1 Hartreset: RESET

Eingehend[14], Nachrichttext: '`RESET`'. Der Effekt ist ein erzwingenes Reset der TRP Skripte. Beispiel LSL Befehl, auszuführen im Treiber:

```
llMessageLinked(LINK_ROOT, -2018160314, "RESET", NULL_KEY);
```

2.6.1.2 Erzwungenes stop: STOP

Bidirectionell[15], Nachrichttext: '`STOP`'. Beendet sofort die Ausführung und entfernt alle RLV Restrictions. Beispielaufruf:

```
llMessageLinked(LINK_THIS, -2018160314, "STOP". NULL_KEY);
```

2.6.1.3 Erzwungenes Start: RUN

Bidirectionell, Nachrichttext: '`RUN`', Schlüssel: Programname zum Starten. Beispielaufruf um das Program '`strawberry`' zu starten:

```
llMessageLinked(LINK_THIS, -2018160314, "RUN". "strawberry");
```

Der Programname berücksichtigt die Schreibweise. Erweitert mit '`.trp`' ergibt den Namen einer Notekarte zum Ausführen, in diesem Fall '`strawberry.trp`'. Die

[14] Der Interpreter kann sie nur empfangen
[15] Der Interpreter kann sie versenden und empfangen

Notekarte ist gestartet, wenn sie vorhanden und ausführbar ist, sonst wird die Nachricht `STOP` versendet.

2.6.2 Interactive Nachrichten

Interactive Nachrichten sind durch das Progrm initiiert und werden via Linknachrichten mit diesen Eigenschaften versendet:

- Das Linkziel ist durch TRP Befehle festgelegt.
- Der Nummerparameter ist beliebig außer -2018160314.

2.6.2.1 Befehlsnachrichten

Eine Befehlsnachricht ist *nichtblokierend*, versendet beim Ausführen des `lm` TRP Befehls, Abschnitt 2.5.2.4. Definition und Beispiel finden sie dort.

```
1 lm <link> <number> <text>
```

Ausführung dieses Befehls führt zum Ausführen diesen LSL Befehls:

```
llMessageLinked(<link>, <number>, "<text>", NULL_KEY);
```

2.6.2.2 Abfragenachrichten

Eine Abfragenachricht ist *blokierend*, versendet beim Auswerten der `lm` Bedingung, Abschnitt 2.5.3.6. Definition und Beispiel finden sie ebenfalls da.

```
1 (if | while) lm <time> <link> <number> <text>
```

Zum Auswerten wird ein Timer gestartet und eine Linknachricht versendet:

```
llSetTimerEvent(<time>);
llMessageLinked(<link>, <number>, "<text>", NULL_KEY);
```

Nun kann zweierlei passieren: Entweder läuft die Zeit ab, dann ist die Bedingung nicht aufgelöst, oder kommt die Antwort per Linknachricht, versendet vom Gerätetreiber via:

```
llMessageLinked(LINK_THIS, <number>, (string)"<resp>", "<key>");
```

Der Wert von `resp` muss eine Zahl sein, auf die die Bedingung ausgewertet wird: 0 wenn sie *wahr* sein soll, 1 falls sie *falsch* werden soll, eine größere Zahl für höhere Fälle. Ist es ein Leerstring, ist die Bedingung nicht auflösbar. Der Wert von `key` ist jeder String außer `NULL_KEY`, etwa ein leerer String.

2.7 Beispieltreiber

An dieser Stelle nehmen wir den Beispieltreiber unter die Lupe, den mitgelieferten Skript ‘waterglass.drv’. Da der Cole etwas länger ist, wird dieser Stückweise erklärt.

2.7.1 Präambel

```
1  //------------------------------------------------------------
2  // $RCSfile: waterglass.drv.lsl,v $
3  //
4  // TRP device driver: Glass of mineral water.
5  //
6  // ->[]    <-2018160314, "RUN", name>
7  // ->[]    <-2018160314, "STOP", NULL_LEY>
8  // ->[]    <0, "drink", NULL_LEY>
9  // ->[]    <0, "isempty?", NULL_KEY>
10 //    []-> <0, "0", "isempty?">
11 //    []-> <0, "1", "isempty?">
12 // ->[]    <0, "detachme", NULL_LEY>
13 // ->[]    <0, "dropme", NULL_LEY>
14 // ->[]    <0, "placeme", NULL_LEY>
15 //
16 //------------------------------------------------------------
17 // Author  Jenna Felton
18 // Version 1.0, $Revision: 1.3 $
19 //         $Date: 2010/09/30 18:09:50 $
20 //------------------------------------------------------------
```

Figure 2.4: Präambel des Gerätetreibers

Den Anfang macht die Präambel, Abbildung 2.4. Hier wird die aktuelle Skript-API untergebracht. Als reiner Kommentar hat dieser Abschnit keine Wirkung als Skript, jedoch ist ihr Niederschreiben wichtig für die verteilte Entwicklung, und zum Erinnern wenn man paar Wochen was anderes getan hat. Die benutzte API-Syntax nutzt der Author seit Jahren schon

Die Spitze Klammer ‘<...>’ notieren eine Linknachricht. Nachrichten die der Skript empfängt, sind nach der Präfix ‘->[]’ angegeben, Nachrichten, die der Skript versendet, stehen nach dem Präfix ‘[]->’. Die drei Komponenten in den Klammern sind der nummerische, String und Schlüsselparameter der Linknachricht. Beispielweise wird die Nachricht ‘<0, "1", NULL_KEY>’ mittels dieses LSL Befehls versendet:

```
llMessageLinked(LINK_THIS, 0, "1", NULL_KEY);
```

Der Rest ist ein wenig CVS, allgemeine Informationen und eine gute Stelle zur Angave von Eänderungen.

2.7.2 Animationsmanagement

2.7.2.1 Warum auslaggern von Animationen?

Die TRP Sprache hat keine Befehle fürs Starten von Animationen oder Schalten von Sound oder Partikeleffekte oder ännliches. Zwei Gründe gibts dafür: Zum Einen macht das die Sprache schlanker. Zum Anderen sind Sachen wie Animationen oder Sound speziell für gegebenes Gerät. Es ist öfter mehr zu tun, als einfach den Avatar trinkend zu animieren. Trinken meint auch das verringern der Menge des Getränks im Glass, d.h. das Animieren ist auch mit dem Verändern des Gerätestatus verbunden.

Das ist der Hauptgrund warum Verwaltung von Animationen (und Sounds oder ännlicher Dinge) die Arbeit des Gerätetreiber ist. Der Interpräter löst diese Arbeit allerdings mittels Versenden von interactiven Nachrichten aus, Abschnitt 2.6.2.

2.7.2.2 Zurück zum Code

Der nächste Codeabschnitt ist die Verwaltung von Animationen und Wassermenge im Glass, Abbildung 2.5.

Das Wasser im Glass ist mittels eines Prims dargestelt, das Prim ist so geformt, dass zum Verringern der Wassermenge es nur die Primhöhe reduzieren muss. Die aktuelle Primhöhe speichert die Variable `fWaterZDim`.

Nun, die Funktion `getDrink()` animiert den Avatar, reduziert den Variablenwert und aktualisiert die Höhe des Waserprim, verringert so den sichtbaren Inhalt im Glass.

Die Funktion `isEmpty()` wird aufgerufen und festzustellen ob das Glass leer ist oder nicht. Die Funktion prüft einfach ob die Variable `fWaterZDim` positiv ist und versendet die Antwort direkt via Linknachricht, die der Interpreter versteht.

Beide Funktionen bieten den Servie, den das Beispielprogram via interaktiven Nachrichten nutzt. Der Code steht auf den Zeilen 14 bis 17, Abbildung 2.3:

Der Befehl "`while not lm 15 this 0 isempty?`" in der Zeile 12 produziert eine Abfragenachricht: Der Interpreter versendet den Befehl '`isempty?`' via Linknachricht und hällt an. Diese Linknachricht wird im Defaultzustand des Treiber empfangen, Abbildung 2.7, und führt zum Aufruf der Funktion `isEmpty()`. Die Funktion versendet die Antwort und der Interpreter entscheidet sich fürs Verlassen oder Weiterführen der Schleife.

Der Befehl "`lm this 0 drink`" in der Zeile 13 produziert eine Befehlsnachricht: Der Interpreter versendet den Befehl '`drink`' ebenfalls via Linknachricht, hält aber nicht hier (der Befehl `wait` ist eine andere Geschichte). Diese Nachricht kommt wieder im Driver an, und führt zum Aufruf der Funktion `getDrink()`. Während der Interpreter wartet, der Avatar führt eine Trinkanimation aus und die Wassermenge im Glass verringert sich.

Die ganze Interaktion wiederholt sich bis das Glass leer ist und die Schleife ist verlassen.

2.7.3 Detachment management

2.7.3.1 Warum auslaggern von Ablege-Aktionen?

Manchmal, wenn man ein Attachment mit TRP aufrüstet, möchte man bestimmtes Code nach dem Ablegen ausführen, das einfachste Beispiel wäre dieses Programabschnitt (Ablegen nur via Interpreter geht nur unter Verwendung von RLV):

```
1 # Ablegen via RLV
2 force detachme
3
4 # Abschluss
5 clean script
```

Warum ist der Code nicht so gut? Es kann nur einer der beiden Befehle erfolgreich ausgeführt werden, entweder das Ablegen oder das Terminieren mir dem Fixieren der Skripte. Beide Befehle auszuführen ist hier nicht möglich.

Eine bessere Lösung besteht darin, den Gerätetreiber es ablegen zu lassen. Die RLV Aktion ist einfach durchs Absenden einer Befehlsnachricht ersetzt und dies schaut dann so aus:

```
1 # Ablegen via Teiber
2 lm this 0 detachme
3
4 # Abschluss
5 clean script
```

Der Treiber wird den Befehl empfangen und kann den Attachment ablegen. Aber forsicht. Tut er das sofort, haben wir nichts am obigen Dilema geändert. Besser sollte der Treiber solange warten, bis er den Programabschluss erkennt. Der Interpreter versendet beim Ausführen des Befehls `clean` auch die Nachricht `STOP`. Erst wenn der Treiber beide Nachrichten empfangen hat, sollte er das Ablegen ausführen.

2.7.3.2 Zurück zum Code

Der dritte Codeabschnitt ist die Verwaltung von Ablegeaktionen, Abbildung 1.6.

Der Code implementiert die oben diskutierte Lösung, behandelt aber etwas mehr Optionen: Dadurch dass auch Linknachrichten dem Lag unterliegen, die Anfrage zum Ablegen kann den Treiber vor oder auch nach der `STOP` Nachricht. Es gibt außerdem drei unterschiedliche Ablegeaktionen, zwei davonführen zusätzliche Aktionen. Das Program löst die Aktionen durch Befehle an den Zeilen 21 bis 33, Abbildung 2.3.

Der Befehl "`lm this 0 detachme`" an der Zeile 22 versendet eine Befehlsnachricht '`detachme`'. Analog, die Befehle an den Zeilen 25, 28 und 30 versenden die Befehlsnachrichten '`dropme`', '`placeme`' und ebenfalls '`detachme`'. Der Abschlussbefehl "`clean script`" an der Zeile 33 versendet die Befehlsnachricht '`STOP`'.

Jede Nachricht wird vom Treiber empfangen und produziert den Aufruf der Funktion '`detachMe()`' mit den Parameterwerten `DA_DETACH`, `DA_DROP`, `DA_PLACE` bzw. `DA_STOP`. Die Funktion legt das Glass nicht sofort, sondern erwartet dass beide Nachrichten ankommen, eine der Ablegenachricht und die Stopnachricht, erst dann wird die Aktion ausgeführt. Im Falle von speziellen Anfragen wird die spezielle Ablegeaktion ausgeführt mit dem Werfen oder Abstellen des leeren Glass inworld.

Wie funktionieren diese Aktionen? Eigentlich können Skripte die Attachments nur ins Invntar ablegen, nicht aber auf den Boden. Um diese Aktion zu emulieren, rezzt der Treiberskript ein Dummmyglass und legt das eigentliche Glass ab. Das Dummy sieht genau aus wie das leere Glass und schließt die erwwünschte Aktion ab.

2.7.4 Defaultzustand

Der letzte Codeabschnitt ist der Defaultzustand, der alle Teile zusammenfügt, Abbildung 2.7. Was passiert nun hier?

Das Erste was der Skript tut ist den Besitzer um Erlaubnis zu fragen. Da es ein Attachment ist, bekommt der Skript das Erlaubnis automatisch: Das Recht, das Attachment abzulegen und das Recht, den Avatar zu animieren. Stellt der Skript den Besitzerwechsel fest, startet der Skript neu um erneut einzufordern.

Die Animation '`hold_R_handgun`' ist eine endlose Animation die das Glass in der Hand hält, sie ist gestartet beim Anlegen und beendet beim Ablegen des Getränkeglass.

Jede angeforderte Aktion ist angefordert via Linknachricht und behandelt in einer der zuvor vorgestelten Funktion, daher macht die `link_message` Ereignisbehandlung nichts anderes als die Anfragen an die richtige Stelle zu delegieren.

2.7.5 Dummyglass

Nun, eigentlich ist der Treibercode vogestellt, es fehlt aber ein kleines Detail: Das Hilfsobjekt. Es schaut genau wie leeres Glass ohne Inhalt (wird ja leer abgelegt). Es wird physikal sein damit es schön fällt und es wird temporär sein um die Gegend nicht mit leerem Geschirr voll zu stellen. Dieses Objekt liegt im Objektinventar des Getränkeglass damit der Treiber es rezzen kann.

Man kann es ja nicht gescriptet lassen, es wird gut fuktionieren, wenn man es schmeißen möchte. Möchte man es dagegen aufstellen, sollte er stehen bleiben, könnte aber umkippen. Nicht schön. Besser es nicht-physikalisch machen und aufstellen, sobald es die Tischoberfläche berührt. Das erledigt ein kleines Skript das in diesem Objekt liegt, Abbildung 2.8.

Während der Initialisierung macht der Skript das Hilfsobjekt temporär und physikalisch, so dass man es nicht vergisst. Wird das Objekt gerezt, bekommt es gesagt ob es geschmissen oder abgestellt wurde, der Skript notiert das. Das Objekt fällt runter und beim Aufschlagen entscheidet sich der Skript ob er das Objekt weiterrollen lässt oder aufstellt und nicht-physikalisch macht.

Wie auch immer, wenn Sie das Beispielglass mit TRP aufrüsten, wie in 2.3.2 beschrieben, müssen Sie sich um diesen Hilfsobjekt nicht zu kümmern, es liegt bereits im Inventar des Beispielglass und hat schon diesem innenliegen.

Das war's schon

Danke sehr fürs Lesen des Benutzerhandbuchs. Jeden Erfolg beim Aufrüsten Ihrer Produkte mit TRP wünsche Ihnen der JFS shop.

```
22 integer WATER_PRIM      = 2;
23 float   WATER_DIAM      = 0.049;
24 float   DRINK_STEP      = 0.015;
25 float   fWaterZDim      = 0.2;
26
27 getDrink() {
28     if (fWaterZDim > 0.0) {
29         llStartAnimation("drink");
30
31         llSleep(1.0);
32
33         fWaterZDim -= DRINK_STEP;
34
35         if (fWaterZDim > 0.0) {
36             llSetLinkPrimitiveParams(WATER_PRIM,
37                 [PRIM_SIZE, <WATER_DIAM, WATER_DIAM, fWaterZDim>]);
38         }
39         else {
40             llSetLinkAlpha(WATER_PRIM, 0.0, ALL_SIDES);
41         }
42     }
43 }
44
45 isEmpty() {
46     if (fWaterZDim > 0.0) {
47         llMessageLinked(LINK_THIS, 0, "1", "isempty?");
48     }
49     else {
50         llMessageLinked(LINK_THIS, 0, "0", "isempty?");
51     }
52 }
```

Figure 2.5: Animationsmanagement

```
54 vector  REZ_OFFSET     = <0.3, -0.3, 0.5>;
55 vector  REZ_VELOCITY   = <0.25, 0.0, 0.0>;
56
57 integer DA_DETACH      = 1;
58 integer DA_DROP        = 2;
59 integer DA_PLACE       = 3;
60 integer DA_STOP        = 4;
61 integer gDetachActn    = FALSE;
62
63 detachMe(integer actn) {
64     if (!gDetachActn) gDetachActn = actn;
65     else {
66         if (actn == DA_STOP) {
67             if (gDetachActn == DA_STOP) return;
68             else                        actn = gDetachActn;
69         }
70         else if (gDetachActn != DA_STOP) return;
71
72         if (actn == DA_DROP || actn == DA_PLACE) {
73             string inv = llGetInventoryName(INVENTORY_OBJECT, 0);
74             if (inv != "") {
75                 rotation rot = llGetRot();
76                 vector   pos = llGetPos();
77                 vector   vel = ZERO_VECTOR;
78
79                 pos = pos + REZ_OFFSET*rot;
80                 if (actn == DA_DROP) vel = REZ_VELOCITY*rot;
81
82                 llRezAtRoot(inv, pos, vel, ZERO_ROTATION, actn);
83
84                 if (llGetInventoryType(inv) == INVENTORY_OBJECT) {
85                     llRemoveInventory(inv);
86                 }
87             }
88         }
89
90         llDetachFromAvatar();
91     }
92 }
```

Figure 2.6: Verwaltung von Ablegeaktionen

```
94  default {
95      state_entry() {
96          llRequestPermissions(llGetOwner(),
97              PERMISSION_ATTACH|PERMISSION_TRIGGER_ANIMATION);
98
99          llSetLinkAlpha(WATER_PRIM, 0.5, ALL_SIDES);
100         llSetLinkPrimitiveParams(WATER_PRIM,
101             [PRIM_SIZE, <WATER_DIAM, WATER_DIAM, fWaterZDim>]);
102     }
103
104     changed(integer change) {
105         if (change & CHANGED_OWNER) llResetScript();
106     }
107
108     run_time_permissions(integer perm) {
109         if(perm|PERMISSION_TRIGGER_ANIMATION) {
110             llStartAnimation("hold_R_handgun");
111         }
112     }
113
114     attach(key id) {
115         if (id == NULL_KEY)    llStopAnimation("hold_R_handgun");
116         else                   llStartAnimation("hold_R_handgun");
117     }
118
119     link_message(integer sender, integer num, string msg, key id) {
120         if (num == 0) {
121             if      (msg == "isempty?")  isEmpty();
122             else if (msg == "drink")     getDrink();
123
124             else if (msg == "detachme")  detachMe(DA_DETACH);
125             else if (msg == "dropme")    detachMe(DA_DROP);
126             else if (msg == "placeme")   detachMe(DA_PLACE);
127         }
128
129         if (num == -2018160314) {
130             if (msg == "RUN")            llResetScript();
131             if (msg == "STOP")           detachMe(DA_STOP);
132         }
133     }
134 }
```

Figure 2.7: Der Defaultzustand

```
//-----------------------------------------------------------
// $RCSfile: dummyglass.lsl,v $
//
// Dummy glass for mineral water.
//
//-----------------------------------------------------------
// Author  Jenna Felton
// Version 1.0, $Revision: 1.1 $
//         $Date: 2010/09/30 13:27:47 $
//-----------------------------------------------------------

integer DA_PLACE     = 3;
integer placed       = FALSE;
key     owner;

default {
    state_entry() {
        llSetPrimitiveParams([
            PRIM_PHYSICS, TRUE,
            PRIM_TEMP_ON_REZ, TRUE]);
    }

    on_rez(integer param) {
        if (param == DA_PLACE) placed = TRUE;
        owner = llGetOwner();
    }

    collision_start(integer num) {
        if (!placed) return;
        while (num-- > 0) {
            if (llDetectedKey(num) == owner) return;
        }
        llSetPrimitiveParams([
            PRIM_PHYSICS, FALSE,
            PRIM_ROTATION, ZERO_ROTATION]);
    }
}
```

Figure 2.8: Script inside help object

Links

[1] The Cool VL Viewer. http://sldev.free.fr/.

[2] Emerald Viewer. http://emeraldviewer.net/.

[3] Imprudence Viewer. http://imprudenceviewer.org/.

[4] Jenna's Second Life (blog). http://jennassl.blogspot.com/2010/05/actual-shop-list.html.

[5] Marine's Sub-space. http://realrestraint.blogspot.com/?zx=8b9d97d8abc05fba.

[6] JFS main store. http://slurl.com/secondlife/2raw/189/94/23.

[7] Phoenix Viewer. http://www.phoenixviewer.com/.

[8] RLV. http://www.erestraint.com/realrestraint/.

[9] LSL Protocol/RestrainedLoveAPI. http://wiki.secondlife.com/wiki/LSL_Protocol/RestrainedLifeAPI.

[10] Rainbow Viewer. http://my.opera.com/boylane/blog/.

[11] Third-Party Viewer Directory. http://viewerdirectory.secondlife.com/.

[12] Policy on Third-Party Viewers. http://secondlife.com/corporate/tpv.php.